GESELLSCHAFT
FÜR INFORMATIK

Deepak Dhungana, Leen Lambers, Leif Bonorden, Sören Henning (Hrsg.)

Software Engineering 2024 - Companion Proceedings (SE-C 2024)

26.2.2024 – 28.2.2024
Linz, Österreich

Gesellschaft für Informatik e.V. (GI)

Inhaltsverzeichnis

Preface

Software Engineering Workshops

21st Workshop on Automotive Software Engineering (ASE'24)

6th Workshop on Avionics Systems and Software Engineering (AvioSE'24)

Workshop Generative and Neurosymbolic AI in Software Engineering (GenSE'2024)

6th Workshop on Software Engineering for Cyber-Physical Production Systems (SECPPS'24)

Student Research Competition

Autor*innenverzeichnis

Preface

Message from the SE'24 Workshop Chairs

Deepak Dhungana[1] and Leen Lambers[2]

Abstract: This volume includes the proceedings of the Workshops of the 2024 Software Engineering conference (SE'24). SE is the leading conference on software engineering in German-speaking countries and is annually organized by the Gesellschaft für Informatik (GI). The SE conference series serves as a platform to exchange experiences and insights in the area of software engineering for which it addresses an audience from both practice and academia. The workshops were held on 26th and 27th of February 2024. The SE'24 was held at the Johannes Kepler University in Linz, Austria.

1 Workshops

The workshops were selected by the workshop chairs, considering the feasibility of the proposed workshop and the potential to attract an engaged audience. All submitted proposals were of high quality and therefore were accepted.

- *5th Workshop on Anforderungsmanagement in Enterprise Systems-Projekten (AESP'24)* Many enterprise systems selection, implementation, and development projects fail due to missing, incorrect, inadequate, or incomplete requirements. This is often because these projects involve incorrect expectations, disagreements in definitions, and differing opinions on requirements management between clients and suppliers. In addition to the requirements driven by enterprise systems, companies involved in enterprise systems projects often face additional organizational requirements and challenges, such as (i) New or modified business processes (ii) New or modified corporate organization (iii) Need for capacity and availability of relevant project personnel, such as key users (iv) ERP and process competence of employees (v) ERP capability of the organization and its personnel (vi) Financing and budgeting – operationalization of agile methods, etc. These challenges were highlighted, discussed, and debated in this workshop. `https://www.sis-consulting.com/se24-anforderungsmanagement-in-enterprise-systems-projekten/`
- *21th Workshop on Automative Software Enineering (ASE'24)* Like its predecessors, the 21st Workshop on Automotive Software Engineering addresses the challenges of software development in the automotive sector, exploring suitable methods, techniques, and tools for this purpose. With increasingly connected vehicles, modern driver assistance functions, and the challenges of fully automated driving, automotive software plays an ever-important role in today's context. In addition to the continuously rising complexity, stricter requirements for reliability, safety (both security and safety),

1 IMC University of Applied Sciences, Krems, Austria, deepak.dhungana@fh-krems.ac.at
2 Brandenburgische Technische Universität Cottbus-Senftenberg, Germany, leen.lambers@b-tu.de

doi:10.18420/sw2024-ws_01

and data protection (privacy) must be met. Furthermore, distraction-free and intuitive multimodal operation of vehicle applications through voice and gesture control is becoming increasingly significant. The trend towards connectivity has already reached vehicles. Thus, driving is being transformed by advancing "digital cultures": value-added services (e.g., social media, streaming, office applications) will be even more seamlessly integrated into vehicles and can be operated by users while driving. This workshop discussed challenges and solution approaches in Automotive Software Engineering, with a particular focus on the use of agile methods in a regulated environment. `https://ase-workshop.github.io/2024/`

- *6th Workshop on Avionics Systems and Software Engineering (AvioSE'24)* Software development in the aerospace domain is driven by demanding fault tolerance, increasing complexity, new application potentials, rising certification effort, and increasing cost pressure. New software development methodologies are required for future applications such as e.g. Advanced Air Mobility (AAM), aircrew (workload) reduction, and further enhancement of existing functionality. At the same time, there are challenges in communication and navigation in airspace, certification for multi-core processors, artificial intelligence (AI) as well as security of software, hardware, and connectivity. The aim of the workshop is to exchange information on software and systems engineering methods and tools with an application in avionics. Presentations of new methods and technologies in this field are welcome. This was a one-day event with presentations, keynotes and discussions. `https://aviose-workshop.github.io/`
- *1st Workshop on Generative and Neurosymbolic AI in Software Engineering (GenSE'24)* Generative methods have strongly influenced developments in the field of Artificial Intelligence (AI) over the past year, ranging from ChatGPT to open-source models like Llama-2. The automatic generation of computer code or structured data based on a description in natural language can be considered the first application of generative models in software development. However, since the accuracy and reliability of the outputs of such generative models cannot be guaranteed, their practical application often comes with risks. In software development, this can lead to software errors or result in security vulnerabilities. In this workshop, we aim to discuss challenges in the use of generative AI methods in software engineering and propose and validate solutions to the aforementioned risks and challenges. The practical application of neurosymbolic approaches will be particularly emphasized. The workshop is targeted at researchers, scientists, developers, and users from both the academic and industrial fields. `https://gense-workshop.github.io/`
- *Quantum Software Engineering Meetup (QSE Meetup)* Quantum computing promises to solve problems beyond the capabilities of classical computing. To harness the potential of emerging quantum hardware, extensive method development within the field of quantum software engineering is required. This involves the need for abstraction concepts, programming languages, compiler technology, testing and analysis methods, processes, and guidelines that allow for the broad operation and efficient utilization of quantum computers. The goal of the QSE-MeetUp is to engage

the software engineering community in the field of quantum computing and to enhance the contribution of software engineering in making quantum computing accessible and applicable. The Meetup included an introductory keynote on quantum computing and a series of keynote speeches addressing challenges in quantum software engineering from both academic and practical perspectives. It was followed by a panel discussion involving speakers and participants. https://tva.kastel.kit.edu/aktivitaeten/Quantum_Software_Engineering_MeetUp_2024.php

- *6th Workshop on Software Engineering for Cyber-Physical Production Systems (SECPPS'24)* Software plays an essential role in operating industrial production systems efficiently. Despite variability and complexity being core challenges in cyber-physical production systems (CPPS), recent developments in software engineering have yet to make significant inroads into the automation of production systems. Various ways to integrate software engineering will be discussed in this workshop. https://rickrabiser.github.io/secpps-ws/se24

2 Acknowledgements

We would like to thank all those who contributed to making the SE'24 workshops possible.

First of all, we would like to thank the workshop organizers for their workshop ideas and the engagement and energy they put into making the workshops a reality. Namely, we thank:

- Christoph Weiss and Johannes Keckeis for organizing the Workshop on Anforderungsmanagement in Enterprise Systems-Projekten (AESP'24)
- Stefan Kugele and Franz Wotawa for organizing the Workshop on Automotive Software Engineering (ASE'24)
- Marina Reich, Björn Annighöfer, and Andreas Schweiger for organizing the Workshop on Avionics Systems and Software Engineering (AvioSE'24)
- Rubén Ruiz-Torrubiano, Alois Haselböck, and Danilo Valerio for organizing the Workshop on Generative und Neurosymbolische KI im Software-Engineering (GenSE'24)
- Ina Schaefer, Michael Felderer, Malte Lochau for organizing the Quantum Software Engineering MeetUp (QSE Meetup '24)
- Sandra Greiner, Jörg Walter and István Koren for organizing the 6th Workshop on Software Engineering for Cyber-Physical Production Systems (SECPPS'24)

Furthermore, we are grateful to the members of the workshop program committees, who reviewed the workshop submissions and ensured the quality of the presented research. Additional thanks go to the authors of all workshop submissions and the attendees of the workshops for making SE'24 an interesting venue.

A special thanks goes to the General Chairs of the SE'23 Rick Rabiser and Manuel Wimmer, the program committee co-chairs Iris Groher and Andreas Wortmann, the industry track co-chairs Stefan Sauer and Alois Zoitl, the publicity chair Judith Michael, the sponsoring chair Reinhold Plösch, the proceedings chair Bianca Wiesmayer, the web chair Daniel Lehner, the local chair Stefan Klikovits, the student volunteer chair Lisa Sonnleithner, the student research competition chairs Leif Bonorden and Sören Henning, the organization and registration chairs Birgit Breitschopf and Ursula Schwarzgruber for their continued and outstanding support. Their work helped the workshop organizers to create a great environment for the workshops.

Finally, we would like to acknowledge the team of the GI Digital Library who made publishing this volume possible, as well as the EasyChair team, whose software was instrumental during the review processes.

Krems, Cottbus, February 2024
Deepak Dhungana, Leen Lambers

Software Engineering Workshops

21st Workshop on Automotive Software Engineering (ASE'24)

Verbesserung der Testqualität mit dem Testing Quality Audit

Ein Erfahrungsbericht

Ralf Reißing[1] Christoph Gomringer[2] Frank Houdek[3]

Abstract: Die Entwicklung der Elektrik/Elektronik (E/E) im Automobil ist typischerweise verteilt auf den Fahrzeughersteller (OEM) und seine Zulieferer. Mitentscheidend für eine hohe Qualität der entwickelten Systeme ist eine hinreichende Absicherung der Erfüllung aller jeweils relevanten Anforderungen durch die beteiligten Parteien. Mercedes-Benz hat ab 2008 das Testing Quality Audit (TQA) eingeführt, um bei Bedarf die Güte der durchgeführten Testaktivitäten bei Zulieferern zu bewerten und zu verbessern. Dieser Beitrag stellt das TQA vor und diskutiert seine Weiterentwicklung über die Jahre seit der Einführung sowie die dabei gemachten Erfahrungen sowohl zum TQA-Ablauf als auch zu typischen Befunden bei den TQAs.

Keywords: Testen; Testprozess; Testqualität

1 Einführung

Das Testing Quality Audit (TQA) hat sich bei Mercedes-Benz ab dem Jahr 2008 etabliert. Gegenstand des TQA sind die Testprozesse, Testaktivitäten und Testartefakte bei Lieferanten von Systemen oder Komponenten, die Software enthalten. Das sind typischerweise einzelne Steuergeräte oder Verbünde aus Steuergeräten, Sensoren und Aktuatoren. Ziel eines TQA ist es, die Effektivität und die Effizienz der Testaktivitäten zu verbessern, wobei sowohl Lieferanten als auch die Entwicklungsbereiche von Mercedes-Benz selbst betrachtet werden. Dazu werden die Aktivitäten im statischen und dynamischen Test analysiert, bewertet und Verbesserungsvorschläge generiert.

Dieser Abschnitt beschreibt die Entstehung, die Grundprinzipien und weitere Entwicklungen des TQA bis heute. Abschnitt 2 zeigt, wie ein typisches TQA abläuft und wie die Ergebnisse aussehen. In Abschnitt 3 geht es um Beobachtungen und Erfahrungen aus TQAs in vielen Projekten, sowohl zur Durchführung als auch zu den Ergebnissen. Abschnitt 4 schließlich fasst zusammen und gibt einen Ausblick.

[1] Hochschule Coburg, Fakultät Maschinenbau und Automobiltechnik, Friedrich-Streib-Str. 2, 96450 Coburg, Germany, ralf.reissing@hs-coburg.de
[2] Mercedes-Benz AG, Research & Development, 71059 Sindelfingen, Germany, christoph.gomringer@mercedes-benz.com
[3] Mercedes-Benz AG, Research & Development, 71059 Sindelfingen, Germany, frank.houdek@mercedes-benz.com

doi:10.18420/sw2024-ws_02

1.1 Entstehung

Im Jahr 2007 erfolgte ein spontaner Lieferantenbesuch durch einen Mitarbeiter der Daimler-Konzernforschung (Bereich Software) bei einem Steuergeräte-Entwicklungsprojekt mit Qualitätsproblemen. Der Auftrag dazu kam aus dem Entwicklungsbereich Pkw. Ziel war es, die Testaktivitäten des Lieferanten im Projekt zu begutachten und die Befunde an den Auftraggeber zu berichten. Ein Vorteil dieser Vorgehensweise war, dass der Gutachter zwar vom selben Unternehmen wie der Auftraggeber kam, aber eine vergleichsweise neutrale Position einnahm, also auch Verbesserungspotential beim Auftraggeber aufzeigen konnte. Aus Sicht des Auftraggebers war die Unternehmung erfolgreich, weil sie dazu beitrug, die Zusammenarbeit mit dem Lieferanten zu verbessern.

Der Erfolg sprach sich im Entwicklungsbereich Pkw herum. Weitere Anfragen zu ähnlichen Begutachtungen bei Lieferanten führten zur Etablierung der internen anstleistung Testing Quality Audit (TQA). Der Name lehnt sich an eine andere, bereits etablierte Dienstleistung an, das Code Quality Audit (CQA). Beim CQA wird die Codequalität, aber auch die Übereinstimmung von Codestruktur und Softwarearchitektur untersucht. Diese Dienstleistung bot ein anderes Team der Konzernforschung an. Der Begriff Audit deutet an, dass es um die Analyse und Bewertung von Prozessen geht. Allerdings ist die Grundlage des TQA kein (inter)nationaler Standard, sondern Kenntnisse der Best Practices im Bereich Test (z.B. aus dem ISTQB Certified Tester [IS23]) und der Besonderheiten der Entwicklungsprozesse bei Mercedes-Benz sowie des zugehörigen Lieferantenmanagements.

Vom CQA wurde die Idee übernommen, mit einer standardisierten Berichtsvorlage zu arbeiten. Zunächst gab es nur einen Endbericht, der die vorgefundene Situation beim Lieferanten ausführlich beschreibt, bewertet (Stärken, Schwächen, Risiken) und Verbesserungsvorschläge (kurz-/mittel-/langfristig) unterbreitet. Da die Erstellung des Endberichts mit 4-6 Wochen den Auftraggebern aber zu lange dauerte, wurde später zusätzlich ein innerhalb einer Woche verfügbarer kurzer Zwischenbericht eingeführt, dessen Inhalte allerdings als vorläufig deklariert sind. Für die Vorbereitung und Durchführung der TQAs wurde eine Checkliste mit vielen Prüfpunkten erarbeitet.

Die Indikation für die Durchführung eines TQA waren anfangs vor allem Lieferanten-Projekte in Schwierigkeiten, typischerweise starke Zeitverzögerungen bei der Lieferung von Produktversionen und/oder mangelnde Qualität der gelieferten Produktversionen. Später gab es auch proaktive TQA, um beispielsweise die Fähigkeiten neuer Lieferanten vor Vergabe eines Entwicklungsauftrags zu beurteilen. Diese wurden allerdings wieder eingestellt, da kein reales Projekt mit realen Artefakten beurteilt werden konnte, sondern nur Standardprozesse, deren konkrete Umsetzung in Projekten aber theoretisch blieb. In seltenen Fällen gab es auch auch unternehmensinterne TQA bei Eigenentwicklungen von Software-Modulen oder ausführbaren Modellen, die später von Lieferanten in deren Steuergeräte integriert werden sollten.

1.2 Grundsätze

Bei der Definition von Ablauf und Inhalt eines TQA waren die folgenden Grundsätze handlungsleitend:

- Das TQA betrachtet sowohl Effektivität als auch Effizienz der Testaktivitäten – sowohl beim Lieferanten als auch beim beauftragenden Fachbereich.
- Das TQA ist erfahrungsbasiert, d.h. es werden typische Fehler abgeprüft, die organisatorischer, strategischer, planerischer, oder technischer Natur sein können. Intuitive Stichproben in vorhandenen Testartefakten decken oft weitere Befunde auf. Dieses Vorgehen setzt viel Wissen und Erfahrung bei den Gutachtern voraus.
- Das TQA ist firmenspezifisch, d.h. das Vorgehen des Lieferanten muss zu den Entwicklungsprozessen des Auftraggebers passen, z.B. sind spezielle Anforderungen an durchzuführende Testaktivitäten beim Lieferanten zu berücksichtigen. Aber auch die Betrachtung der Anforderungsdokumente, der Testabläufe und Testumgebungen beim Auftraggeber und deren Schnittstellen zum Lieferanten (z.B. gemeldete Fehler, Fehlernachtests) ist relevant.
- Das TQA ist lösungsorientiert, d.h. die vom TQA-Team gemachten Verbesserungsvorschläge sind Empfehlungen an den Lieferanten und den Auftraggeber. Sind sind nicht verbindlich und es wird vom TQA-Team nicht überprüft, ob sie umgesetzt wurden. Bei einem zweiten TQA im selben Projekt (ein sogenanntes Delta-TQA) wird allerdings zu Beginn abgefragt, wie mit den Befunden und Empfehlungen aus dem ersten TQA umgegangen wurde.
- Die Anforderungen anwendbarer Standards wie ASPICE (ASPICE, [VD17]), ISO 26262 [IS18] und von Mercedes-Benz-Normen an die Testaktivitäten werden berücksichtigt.
- Die Bewertung erfolgt in mehreren Dimensionen (siehe Abschnitt 1.3), was für die Beurteilung wichtiger ist als die daraus abgeleitete, eher artifizielle Gesamtnote.
- Das TQA-Meeting dauert maximal einen Arbeitstag, um die Personalressourcen beim Lieferanten nicht übermäßig in Anspruch zu nehmen.

Oft wird die Frage gestellt, wie sich das TQA von einem ASPICE-Assessment abgrenzt. Beide betrachten ja die Qualität der Testaktivitäten in einem Projekt. Ein wichtiges Unterscheidungsmerkmal ist die Berücksichtigung der firmenspezifischen Verhältnisse bei Mercedes-Benz und der projektspezifischen Anforderungen an den Lieferanten. Außerdem liegt der Fokus intensiver auf den Testaktivitäten. Anderen Aktivitäten wie Anforderungs- oder Konfigurationsmanagement werden beim TQA bei Bedarf ergänzend mit einbezogen, während ein ASPICE-Assessment alle Entwicklungsaktivitäten gleichermaßen in den Blick nehmen muss. Dadurch bleibt im TQA mehr Zeit für tiefergehende Untersuchungen der Testaktivitäten, Stichproben in den zugehörigen Testartefakten und der Besichtigung

sowie Demonstration der eingesetzten Testumgebungen. Für den Lieferanten wertvoll sind insbesondere die Empfehlungen aus dem TQA, denn ein ASPICE-Assessment liefert in der Regel wenig konkrete Verbesserungsvorschläge für aufgedeckte Kritikpunkte. Es ist von Vorteil, wenn vor dem TQA bereits ein ASPICE-Assessment stattgefunden hat, da das TQA-Team dann in der Vorbereitung besser beurteilen kann, wo eine vertiefte Begutachtung der Testaktivitäten besonders lohnenswert ist.

1.3 Bewertung

Für die Bewertung des im TQA untersuchten Projekts haben sich die folgenden Bewertungsbereiche etabliert, die im Folgenden kurz charakterisiert werden:

- Testmanagement: Organisation von Qualitätssicherung und Testen, Zeit- und Ressourcenplanung, Qualifikation Testmanager, Traceability, Erhebung von KPIs
- Testvorgehen: Teststrategie, Prozesse, Methoden, Werkzeugeinsatz, konform zu anwendbaren Standards
- Testspezifikation: Testfallermittlung, Testfalldokumentation, Testüberdeckung
- Testberichte: Form und Inhalt der Testberichte
- Testtechnologie: Eignung der eingesetzten Werkzeuge und Testumgebungen
- Fehlermanagement: Organisation und Prozess der Fehlerdokumentation und der Fehlerabstellung, Austausch von Fehlern und Statusupdates zwischen allen beteiligten Parteien, Fehlerstatistiken

Abbildung 1 zeigt, wie sich diese Bereiche in der Gesamtbewertung in der Management Summary des Endberichts wiederfinden. Die Bewertungskala sind hier deutsche Schulnoten (1 = sehr gut bis 6 = ungenügend) mit Zwischenschritten von 0,5. Es wird pro Bereich jeweils über alle betrachteten Teststufen gemittelt.

1.4 Weitere Entwicklung

Das TQA verbreitete sich vom Pkw-Bereich aus auch in die anderen Bereiche wie Truck, Van und Bus, die ebenfalls Lieferanten mit Qualitätsproblemen hatten. Durch die zunehmende Eigenentwicklung strategisch wichtiger Steuergeräte und Softwareumfänge bei Mercedes-Benz gibt es zunehmend auch interne TQA, die solche Projekte in den Fokus nehmen.

Die zunehmende Etablierung von ASPICE bei den Lieferanten führte zu einer Anpassungen in der Vorgehensweise: Die Benennung der Teststufen in den Agenda-Punkten orientiert sich inzwischen an den Namen von ASPICE (also zunächst ENG.x, später dann SYS.x bzw.

Kriterium	Bewertung
Testmanagement	2,5
Testvorgehen	3
Testspezifikation	3
Testberichte	2
Testtechnologie	2,5
Fehlermanagement	2
Gesamt	2,5

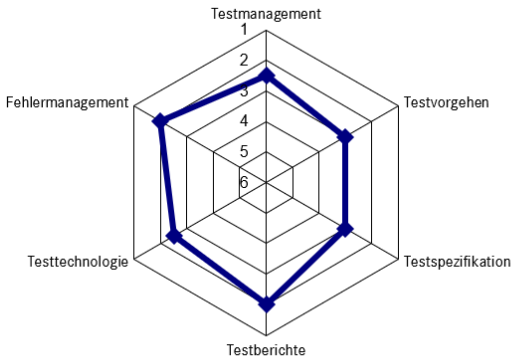


Abb. 1: Darstellung der Bewertungsdimensionen im Bericht

SWE.x). Zunächst war die Benennung an das Glossar des ISTQB angelehnt gewesen. Beim Bewertungsschema wurde von einer eigenen Skala (0-3, 0 = gar nicht, 3 = vollständig) auf das vertrautere deutsche Notenschema (1-6) umgestellt, damit der beauftragende Fachbereich die Bewertungen besser interpretieren konnte.

Wegen steigender Nachfrage nach TQA und daher höherer Arbeitslast beim TQA-Team bei der Durchführung und Dokumentation der Ergebnisse ist seit Jahren immer ein externer Gutachter dabei. Der externe Gutachter ist mit den Abläufen bei Mercedes-Benz zwar vertraut, arbeitet aber nicht dort. Neben der Entlastung bei den Dokumentationsaufgaben hat der externe Gutachter daher auch den Vorteil, andere Perspektiven einzubringen und weniger betriebsblind für Optimierungspotenziale bei Mercedes-Benz zu sein.

Die TQA-Methodik wird alle paar Jahre auf den Prüfstand gestellt und angepasst bzw. optimiert. Bspw. wurde zuletzt 2022 die Vorlage des Endberichts überarbeitet, um die Klarheit der Darstellung zu erhöhen, aber auch um den Aufwand der Erstellung zu reduzieren. Kleinere Optimierungen finden jederzeit statt.

2 Ablauf eines TQA

Der Ablauf eines TQA gliedert sich in drei Phasen: Vorbereitung, Durchführung und Bericht. Diese Phasen werden im Folgenden genauer betrachtet.

2.1 Vorbereitung

Die Entscheidung ein TQA für ein bestimmtes Projekt durchzuführen, basiert auf mehreren Kriterien, welche vorab mit dem beauftragenden Fachbereich besprochen werden. Hierbei gilt es zeitliche Aspekte (aktueller Projektzeitplan, weitere Audits/Assessments) sowie die Kritikalität (aktueller Projektstatus, Erfahrung Zulieferer, Komplexität Technologie) zu berücksichtigen.

Ein weiterer Bestandteil der Vorbereitung ist es, die Ziele des TQAs und den Rahmen (Projektgegenstand, Testobjekte) klarzustellen. Gerade in komplexeren Projekten mit vielen Stakeholdern sollte das Projekt und die zugehörigen Testobjekte klar definiert und abgegrenzt sein. Auf Basis dieser Informationen wird dann die Agenda des TQA erstellt. Es wird zudem geklärt, welche Teststufen im Projekt tatsächlich vorhanden sind und welche Personen aus dem Projekt bei den einzelnen Agendapunkte anwesend sein sollten.

Die inhaltliche Vorbereitung der Gutachter profitiert von vorab bereitgestellten Dokumenten wie Projektplan, Organigramm, Testkonzepten, aktuellen Fehlerberichten und schon vorhandenen ASPICE-Assessment-Ergebnissen. Für das Projekt steht die abgestimmte Agenda als Anhaltspunkt für die Vorbereitung zur Verfügung. Das Projekt muss für das TQA keine neuen Dokumente erstellen. Eine kurze Projekt- und Teamübersicht sowie das Testkonzept sind aber für den Einstieg in das TQA-Meeting sehr hilfreich. Die weiteren Inhalte wie Anforderungen, Testspezifikationen, Implementierung, Fehler und Kennzahlen (KPIs) sollen direkt in den verwendeten Tools oder Testberichten demonstriert werden. Dadurch soll nicht nur der Vorbereitungsaufwand beim Projekt gering gehalten werden, sondern auch der Blick auf die realen Arbeitsstände und Vorgehensweisen gewährleistet werden. So bekommt man im TQA-Meeting auch einen Eindruck, wie gut sich die Teammitglieder in der Dokumentenlandschaft des Projekts auskennen.

2.2 Durchführung

In Abhängigkeit des Formats und Standorts, wird das TQA vor Ort (on-site) beim Projektteam oder als Videokonferenz (remote) durchgeführt. Bei den meisten Projekten findet ein on-site-TQA an einem vollen Arbeitstag statt. Bei remote-TQAs in stark unterschiedlichen Zeitzonen bietet es sich aber an, diese auf zwei halbe Tage zu verteilen. Im Gegensatz zu ASPICE-Assessments werden alle Agendapunkte zunächst durchgängig besprochen. Die Konsolidierung der Gutachter erfolgt dann im Nachgang des TQA-Meetings. Für die einzelnen Agendapunkte müssen jeweils nur die zuständigen Personen anwesend sein, typischerweise sind das Testmanager und Tester. Auch das soll den zeitlichen Aufwand beim Projektteam möglichst gering halten.

Es hat sich als sinnvoll erwiesen, als ersten Agendapunkt die Projektorganisation inklusive der Qualitätsmanagement-Aktivitäten zu besprechen. Aufbauend darauf gewinnt man einen ersten Überblick über die Teststufen im Projekt und die allgemeine Teststrategie.

Üblicherweise kann die Teststrategie bereits anhand der Dokumentation im Testkonzept besprochen werden. Anhand der Dokumentation und weiterer Fragen sollten sich hier die Definition und Abgrenzung der Testobjekte (mit Varianten), die Verantwortlichkeiten, sowie die Testziele und verwendeten Teststufen herausstellen.

Anschließend folgt die Durchsprache der einzelnen Teststufen. Die Agenda orientiert sich in diesen Agendapunkten an den üblichen Prozessschritten der Testfallermittlung, Testspezifikation, Testimplementierung, Testdurchführung und Auswertung sowie Berichten der Testergebnisse. Für die Integrationsstufen wird zudem die Integrationsstrategie genauer betrachtet. Liegen vorab bereits Ergebnisse eines ASPICE-Assessments vor, werden diese von den Gutachtern berücksichtigt, um Zeit zu sparen. Meist können die detaillierten Prozessbeschreibungen kurz gehalten und der Fokus auf das "wie wird getestet" gelegt werden. Dabei wird sowohl die Umsetzung als auch die Angemessenheit betrachtet. Im Vergleich zu ASPICE-Assessments werden zudem auch inhaltliche Themen tiefergehend diskutiert und z.B. die Umsetzung von Fault-Injection-Tests in einzelnen Testumgebungen genauer betrachtet oder der Umgang mit Codierrichtlinien besprochen. Zudem werden die verwendeten Testumgebungen genauer betrachtet, nach Möglichkeit vor Ort im Labor. Fragestellungen sind hier bspw. die ausreichende Verfügbarkeit von Testumgebungen oder die Reproduzierbarkeit von Testergebnissen zwischen Testumgebungen.

Ein weiterer Agendapunkt beinhaltet das Fehlermanagement und soll einen Einblick in den aktuellen Status, die Planung der Fehlerbehebung sowie die Zusammenarbeit zwischen den Stakeholdern im Projekt und mit dem Fachbereich beim OEM liefern. Abgeschlossen wird das TQA mit einem kurzen Feedback der Gutachter, um dem Projektteam eine erste Einschätzung zu geben und die weiteren Schritte zu besprechen. Für die Konsolidierung der Einschätzungen der Gutachter werden im Nachgang die gezeigten und ggf. weitere Dokumente als freiwillige Bereitstellung angefragt. Bereitgestellte Dokumente werden grundsätzlich nicht an den Fachbereich weitergegeben. Auf Basis der betrachteten Inhalte und bereitgestellter Dokumente beginnt dann die Erstellung des Berichts.

2.3 Bericht

Beim Berichtswesen über ein TQA entstand zu Beginn nur der Vollbericht. Aufgrund des Umfangs und einer Abstimmung mit dem Lieferanten dauerte es immer einige Wochen, bis dieser vorlag. Um schneller eine erste Rückmeldung geben zu können, wurde mittlerweile ein Vorabbericht eingeführt.

Der Bericht erfolgt üblicherweise in zwei Schritten. Eine erste kurzfristige Rückmeldung in Form eines Zwischenberichts mit Management Summary mit den wichtigstens Eckdaten und Erkenntnissen, sowie einer vorläufigen Bewertung. Der Zwischenbericht ist ein Foliensatz, welcher aus einem Management Summary (eine Folie) sowie einigen Folien mit den wesentlichen Stärken, Schwächen, Risiken und Empfehlungen besteht – vorbehaltlich Anpassungen, welche sich im Rahmen der Erstellung des Endberichts ergeben. Vor allem

das Management Summary auf einer Seite wird intensiv nachgefragt und mittlerweile ca. eine Woche nach dem TQA an das zuständige Management bei Mercedes-Benz verteilt. Der Zwischenbericht geht nicht an den Lieferanten.

Anschließend folgt die Konsolidierung im Detail und die Ausarbeitung des Endberichts. Hierbei wird für jede Teststufe zusammengefasst, was im Audit gezeigt und bewertet wurde. Daraus abgeleitet werden dann Stärken, Schwächen, Risiken und Empfehlungen genannt und erläutert. Die kurzfristigen, mittelfristigen und langfristigen Empfehlungen bilden den Kern des Berichts und sollen einen konkreten, für das Projekt umsetzbaren Mehrwert bieten.

Bevor der Bericht allen Stakeholdern im Fachbereich inkl. höherem Management bereitgestellt wird, bekommt der Lieferant den Endbericht zum Review. Der Lieferant kann den Bericht kommentieren, beispielsweise um auf Missverständnisse der Gutachter hinzuweisen oder eine andere Sichtweise darzustellen. Erst nach Berücksichtigung dieser Rückmeldungen wird der Endbericht finalisiert und bei Mercedes-Benz verteilt.

Das TQA ist mit Bereitstellung des Endberichts vollständig abgeschlossen. Die Verfolgung der empfohlenen Maßnahmen obliegt dem Projektteam beim Lieferanten in Abstimmung mit dem Fachbereich. Bei Bedarf kann nach ca. 6-12 Monaten ein Delta-TQA durchgeführt werden, um den Fortschritt und die Umsetzung der empfohlenen Maßnahmen nochmals genauer zu bewerten.

3 Erfahrungen

3.1 TQA-Prozess

Bei der Durchsprache der Teststufen im TQA-Meeting wurden zwei Ansätze ausprobiert: Bottom-up, d.h. die Teststufen werden in logischer Reihenfolge vom Software-Unittest bis zum Systemtest durchgesprochen, und Top-Down, d.h. die Teststufen wurden beginnend mit der höchsten Teststufe (z.B. Systemtest) bis zu niedrigsten, also Software-Unittest durchgesprochen. Da aufgrund des festen Zeitrahmens das Risiko besteht, nicht alle Teststufen in der verfügbaren Zeit durchzusprechen, hat der Top-Down-Ansatz den Vorteil, dass die für den OEM relevantesten Teststufen in jedem Fall ausführlich besprochen wurden. Aus Sicht der Gutachter hat der Bottom-Up-Ansatz aber den Vorteil, dass die Testvorgehensweise hier deutlich besser nachvollzogen werden kann und insbesondere klar wird, welche Aspekte auf einer Teststufe abgesichert sind und welche Aspekte für spätere Teststufen noch offen bleiben.

In der Anfangsphase wurden TQAs immer vor Ort beim Lieferanten durchgeführt. Nur in wenigen Ausnahmefällen wurde ein TQA remote durchgeführt. Dies war immer dann der Fall, wenn das TQA mit hohen Reiseaufwänden und möglicherweise höheren Gesundheitsrisiken verbunden war. Remote-TQAs haben unter anderem den Nachteil, dass wichtige Indikationen, wie das Verhalten der Personen vor Ort (wie gehen sie miteinander um) weniger gut

beobachtet werden können. Zudem entfallen die Besichtigungen und Live-Vorführungen an den Prüfständen, und damit auch ein Einblick in die operative Kompetenz der Tester.

Im Zuge der Corona-Pandemie ging die Anzahl der TQAs generell zurück, und in den wenigen Fällen, in denen dennoch ein TQA durchgeführt wurde, war dies aufgrund der geltenden Vorschriften immer remote. Mit Wegfall des Remote-Zwangs ist der Anteil der On-Site-TQA wieder deutlich gestiegen. Dennoch ist der Anteil der Remote-TQAs nun höher als vor der Covid19-Pandemie. Gründe hierfür sind zum einen die positive Erfahrungen mit Remote-TQAs (z.B. der Wegfall der Reisezeiten). Viele TQAs sind mittlerweile aber auch Hybrid-TQAs, weil viele Lieferanten ihre Entwickler weltweit verteilt haben, und es oft nicht sinnvoll ist, beispielsweise einen Kollegen aus dem Software-Unittest für nur einen Agendapunkt um die halbe Welt reisen zu lassen. Das hybride TQA-Meeting findet immer noch vor Ort statt, aber entfernte Entwicklungs- und Testteams sind per Videokonferenz zugeschaltet.

Wenngleich ein TQA keine Aktivität ist, bei der die Erfüllung einer Norm überprüft wird, ist zu beobachten, dass es vermehrt TQA-Anfragen aus dem unabhängigen QM-Bereich bei Mercedes-Benz gibt, der die Entwicklungsprojekte begleitet. Der Treiber eines TQAs ist in solchen Fällen oft nicht der Eindruck, dass Defizite beim Testen auf Lieferantenseite vorliegen, sondern der Wunsch, alle zur Verfügung stehenden Audits und Bewertungen beim Lieferanten anzuwenden. Aufgrund der eingeschränkten Verfügbarkeit von Gutachtern ist die Anzahl an TQAs, die pro Jahr durchgeführt werden können, aber beschränkt. Wenn ein Steuergerät also weder eine zentralle Rolle in einer Baureihe hat noch Hinweise auf Defizite im Testen vorliegen, werden solche TQAs normalerweise nicht durchgeführt.

Der Bedarf an TQAs pro Jahr hängt zusammen mit dem Aktivitäten des OEM in der Fahrzeugentwicklung. In manchen Baureihen ist die Anzahl an Neuentwicklungen höher als in anderen Baureihen, bei denen mehr Steuergeräte aus anderen Baureihen übernommen werden. Abbildung 2 zeigt die Anzahl der durchgeführten TQAs pro Jahr. Das Anwachsen in den ersten Jahren ist durch die Einführung bedingt. Die Schwankungen in den Jahren 2011 bis 2019 erklärt sich durch die wechselnden Bedarfe in den Baureihen. Der Einbruch in den frühen 2020er Jahren geht auf die Covid19-Pandemie zurück.

Als optimaler Zeitpunkt für die Durchführung eines TQA hat sich die Phase herausgestellt, in der der Lieferant bereits Produktversionen an den OEM liefert, also bereits die Testprozesse operativ lebt. Gleichzeitig muss die verbleibende Projektzeit aber noch lange genug sein, um die im TQA identifizierten Empfehlungen auch noch umsetzen zu können. Vereinzelt gab es natürlich Abweichungen von diesem Zeitpunkt. Wenn ein TQA zu früh angesetzt wurde, konnten oft nur Pläne auf Folien-Ebene durchgesprochen werden. Wenn ein TQA zu spät angesetzt wurde, konnten Empfehlungen oft nur noch in Folgeprojekten, aber nicht mehr im Projekt selbst umgesetzt werden. Da ein konkretes Projekt betrachtet wird, sind die erarbeiteten Empfehlungen auch erst einmal nur für das konkrete Projekte nutzbar. Selbst andere Mercedes-Benz Projekte beim selben Lieferanten unterscheiden sich meist

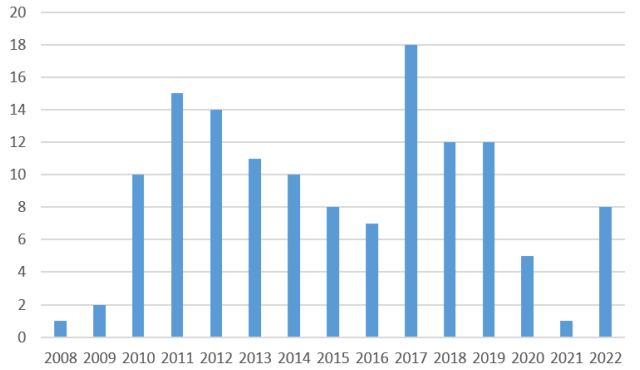


Abb. 2: Anzahl der durchgeführten TQAs pro Jahr

deutlich, so dass die Übertragbarkeit nicht gegeben ist. Die typische Entwicklungsphase der betrachteten Projekte beträgt 3 bis 4 Jahre.

3.2 TQA-Ergebnisse

Es wurden mittlerweile mehr als 130 TQAs über einen Zeitraum von 15 Jahren durchgeführt. Dabei wurden auch einige Veränderungen im Testing über die Zeit beobachtet:

- Nicht zuletzt aufgrund der vermehrten Durchführung von ASPICE-Assessments ist zu beobachten, dass sich grundlegende Themen, wie die Verfügbarkeit notwendiger Dokumente (z.B. Testpläne oder Testkonzepte), aber auch die Tracebility zwischen den Requirements-Ebenen als auch zwischen Requirements und Testfällen verbessert haben. Dies verhindert jedoch nicht, dass es immer noch Lücken gibt oder auch strategische Fehler gemacht werden. Diese treten beispielsweise dort auf, wo ASPICE keine harten Vorgaben macht. Ein Beispiel: ASPICE fordert Testkonzepte für die Teststufen ein. Wir konnten in einem TQA beobachten, dass ein Lieferant für seine neun Teststufen neun verschiedene, nicht aufeinander abgestimmte Testkonzepte erstellt hatte. Hinweise, dass diese doch abgestimmt und miteinander verknüpft sein sollten, wurden mit dem Hinweis, dass ASPICE dies nicht fordere, zurückgewiesen.
- Es ist zu beobachten, dass der Trend eindeutig in Richtung agile Software-Entwicklung geht. Damit verbunden ist häufig die Forderung des OEM, Software-Stände in höherer Taktung zu bekommen. Dies führt zu einem deutlichen Anwachsen der Testlast auf Seiten des Lieferanten. Diese Last kann nicht immer bewältigt werden, mit der Konsequenz dass ungetestete oder wenig getestete Stände beim OEM abgeliefert werden, um die Abgabetermine halten zu können. Und selbst bei 100% Testautomatisierung,

welche selten technisch und wirtschaftlich erreichbar ist, müssen ja die erzeugten Berichte und Befunde beim Lieferanten noch von Hand analysiert werden.

- Bei Software-Unittests konnten immer wieder fragwürdige Vorgehensweisen beobachten werden: So gibt es Lieferanten, die die Erstellung von Testfälle für Software-Unittest an Unterlieferanten auslagern. Die Tests werden auch durchgeführt und zur Bewertung der Codeüberdeckung herangezogen. Allerdings werden für die Testfälle keine Sollresultate aus den Anforderungen abgeleitet, sondern der Test wird entweder immer als bestanden angesehen oder es werden wenige beobachtete Outputs zu Soll-Ergebnissen erklärt. Somit weisen diese Tests bei der Erstdurchführung nur nach, dass es keinen toten Code gibt und die Testdurchführung nicht zu Abstürzen oder Speicherüberläufen führt. Eine andere Schwäche entsteht, wenn Testfälle für Software-Unittest aus sehr detaillierten Architekturbeschreibungen (z.B. detaillierte Flussdiagrammen) abgeleitet werden. Mit diesen Tests werden fast nie Fehler entdeckt – was nicht überrascht, da die sehr detaillierte Architekturbeschreibung auch die Vorgabe für den Implementierer ist. Über diesen Testansatz können nur Fehler im Falle von Übertragungsfehlern von Flussdiagramm zu Code bzw. Flussdisagramm zu Testfall ermittelt werden. Sinnvoller ist hier die Ableitung der Testfälle aus den Anforderungen an die Software-Units.
- Viele Lieferanten versuchen, möglichst große Teile ihrer Testaktivitäten zu automatisieren. Oft ist dies sinvoll. Aber im Rahmen der TQAs konnten auch immer wieder fragwürdige Vorgehensweisen identifiziert werden. Beispielsweise generierte ein Lieferant aus seinen Matlab-Modellen über einen komplexen Prozess Testfälle, die er gegen den aus den Matlab-Modellen automatisch generierten Code laufen ließ. Am Ende konnte über diese Tests damit im Grunde nur nachgewiesen werden, dass über die Codegenerierung keine Fehler entstanden sind. Logische Fehler im Modell wurden natürlich sowohl in den Code als auch in die Testfälle übernommen. Der Prozess selbst war aber so komplex, dass diese Einschränkungen selbst von vielen Mitarbeitern beim Lieferanten nicht erkannt wurden.
- Ein wesentliches Element eines TQA ist immer auch die Einschätzung der Angemessenheit der gewählten Vorgehensweisen. Normen treffen generelle Aussagen. Die Spiegelung an den konkreten Bedingungen eines Projektes sind aber mitzuberücksichtigen. Projektgröße, örtliche Verteilung, Charakteristiken des zu entwickelnden Produktes – all diese Faktoren und noch einige mehr beeinflussen die Sinnhaftigkeit und Ausgestaltung einzelner Testaktivitäten. Exemplarisch sei die Testumgebung für Software-Unittests genannt. Wann ist es sinnvoll, diese Tests bereits auf der Target-Hardware durchzuführen, und wann ist eine Software-in-the-Loop Umgebung geeigneter? Im Rahmen eines TQAs wird insbesondere auch die Angemessenheit der Vorgehensweisen im konkreten Projekt bewertet.

In dem Nachbesprechungen zu den TQAs wird dem TQA-Team immer wieder von den Lieferanten zurückgemeldet, dass sich ein TQA deutlich anders anfühlt als ein klassisches

Audit oder Assessment. Vor allem die zusätzliche Betrachtung der Angemessenheit, aber auch die konkreten Empfehlungen werden durchgehend begrüßt.

4 Zusammenfassung und Ausblick

Das TQA hat sich über 15 Jahre als wichtiger Baustein für die Verbesserung der Zusammenarbeit von Mercedes-Benz mit seinen Zulieferern (intern und extern) im Bereich des Testens etabliert. Es konnte dazu beitragen, konkrete Verbesserungen von Effektivität und Effizienz der Testaktivitäten in Entwicklungsprojekten zu erreichen, sowohl bei den Projekten bei den Lieferanten als auch bei Mercedes-Benz.

Die Erfahrung aus den durchgeführten TQAs zeigt, dass sich Testprozesse zwar besser etabliert haben, aber Projekte auch zunehmend komplexer werden. Eine Herausforderung ist, die für ein Endprodukt benötigten Testaktivitäten zu identifizieren und sinnvoll auf die Bestandteile herunterzubrechen. Diese übergreifende Testdurchgängigkeit und -vollständigkeit wird leider weder durch ASPICE noch durch andere gängige Standards sichergestellt. Es gibt daher erste Überlegungen, das TQA um solche Fragestellungen zu erweitern und z.B. die Verantwortungsmatrix im Bereich des Testens auf Vollständigkeit zu prüfen.

Durch den vermehrten Einzug der ASPICE-Assessments werden Projekte bereits in frühen Phasen auf die grundlegenden Prozesse, Arbeitsprodukte und Verfolgbarkeit hingewiesen. Das hat nicht nur auf die inhaltliche, sondern auch auf die zeitliche Planung des TQAs einen direkten Einfluss. Der Zeitpunkt für die Durchführung eines TQA wird daher zukünftig stärker von geplanten ASPICE-Assessments abhängen und tendenziell in etwas späteren Projektphasen stattfinden, bei welchen die Teststrategie und ihre konkreten Umsetzung schon reifer sind.

Die Bewertungsskala beim TQA muss genügend Spielraum zur Differenzierung der Aspekte und Projekte bieten, gleichzeitig aber auch verständlich und nachvollziehbar sein. Die bisherige Bewertung auf Basis der deutschen Schulnoten hat sich hierbei nicht vollständig bewährt, vor allem da die negativen Extremwerte der Skala nur bei äußerst gravierenden Testlücken in Betracht gezogen werden und der tatsächlich verwendete Wertebereich somit stark eingeschränkt ist. Es bleibt also eine Aufgabe, hier eine noch bessere Lösung zu finden.

Literatur

[IS18] ISO: ISO 26262 – Road vehicles – functional safety, 2018.

[IS23] ISTQB: Our Certifications, 2023, URL: https://www.istqb.org/certifications/certification-list, Stand: 15. 11. 2023.

[VD17] VDA: Automotive SPICE Process Reference Model and Process Assessment Model Version 3.1, 2017, URL: https://automotivespice.com/fileadmin/software-download/AutomotiveSPICE_PAM_31.pdf, Stand: 17. 11. 2017.

Mode Management in Contract-Based Design

Janis Kröger[1], Martin Fränzle[1]

Abstract: Nowadays, safety-critical systems are structured into several operating modes due to their various functionality. To evade inconsistent states in the specification and design, it is essential that these modes and their mode transitions are well defined. This entails a significant effort. This paper proposes an approach to coordinate mode changes between different components using a mode manager. Natural language patterns are designed to reduce the complexity of specifying mode changes. An example system ACC is used to illustrate the concept and patterns.

Keywords: mode management; operating modes; contract-based design

1 Introduction

Safety-critical systems operate in different environments and have various functionalities. An established method for expressing different functionality is the use of modes [Be08] which can also be used as a safety mechanism [Ka15]. Different functionalities, different environments, and ever-increasing safety requirements makes systems complex. To deal with this complexity at design time, the formal verification method of contract-based design has been introduced [SVDP12, Be18], where requirements are specified in the form of contracts. A contract consists of a pair of assumptions (A) and guarantees (G). In the assumptions, the behavior of a valid environment, e.g. an expected input, is specified. The behavior of the component under the valid environment is specified in the guarantees. By using modes in the guarantees, the system behavior becomes mode-dependent, and by using modes in the assumptions, the verification effort can be reduced because some inputs are only required in individual modes [Kr23]. In general, components change their mode only under certain conditions, e.g., at the request, in the occurrences of an error, or when the environment changes. These mode changes confront us with a non-trivial challenge in the specification, like e.g. coordination, consistency and correctness.

Components often rely on each other by responding to the outputs of previous components. In contract specification, the outputs of a component are specified by guarantees, and the inputs are specified by assumptions. If the specified guarantees of a component do not match with the assumption of a following component, we get an inconsistency in the specification. If both the guarantees of the first component and the assumptions of the following component are mode-dependent, it is necessary that the second component knows about a mode change

[1] Carl von Ossietzky Universität Oldenburg, Department für Informatik, Ammerländer Heerstraße 114-118, 26129 Oldenburg, Germany {janis.kroeger,martin.fraenzle}@uni-oldenburg.de

doi:10.18420/sw2024-ws_03

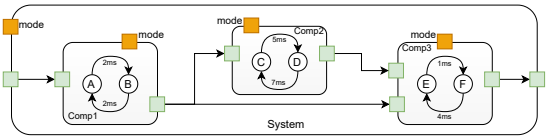


Fig. 1: System with components and different modes and transition times

of the first component. This allows it to adopt the assumptions about the behavior of the first component to avoid a mismatch and an inconsistency in the specification at any time.

Mode transitions take time which varies from component to component and from mode to mode. For example, the system in Figure 1 consists of the components Comp1 to Comp3 with different modes. The modes and the transition times are illustrated as automata in the components. Comp1 receives the inputs of the system, the outputs of Comp1 go into Comp2 and Comp3 as inputs. Comp3 also receives the outputs from Comp2 and provides outputs to the system. Each component is specified by a contract and has mode-dependent assumptions and guarantees. When Comp1 changes its mode, it must inform Comp2 and Comp3 so that they also change mode. Since Comp2 and Comp3 need different times to change modes and adapt, the information from Comp1 must be provided at different times. In larger systems, this can quickly increase the specification size and complexity. In addition, the development of these components requires knowledge of the internal behavior of other components, which contradicts the principles of contract-based design.

One way to reduce the complexity of the specification is the introduction of a so-called mode management component for coordinating the mode changes, as already required in ISO/TR4804 [IS20]. The centralization of mode changes removes the responsibility for the correct initiation of mode changes from the individual components, thereby reducing the specification and encouraging further independent development of the components. This also solves the question in which mode the higher-level system is. This must be clearly defined regarding mode combination of the internal components. While mode combinations can be specified by a table, a mode manager facilitates to directly integrate the mapping into the specification, so that the mapping can be checked in addition to the existing specifications using a virtual integration test (VIT).

As previously noted, components typically do not change modes randomly, but rather under specific conditions. Up to now, it is only possible to specify these conditions in natural language to a limited degree and with a high specification effort. To simplify the specification and incorporate mode mapping, we rely on the works of [Bö17, Bö19, Kr22, Kr23]. Our contribution introduces two new patterns in natural language contracts to i) express triggers for mode changes based on observable behavior and ii) to express the mapping of modes in the specification. Furthermore, the concept of a mode manager is integrated into an already established contract-based design framework by using existing concepts to centralize mode changes. Since mode changes are limited in time, we focus on their timing properties.

2 Related Work

Many temporal logics such as Linear Temporal Logic (LTL) [Pn77], Metric Temporal Logic (MTL) [Ko90], and Metric Interval Temporal Logic (MITL) [AFH96] are the basis of specification to express timing behavior of systems. They differ mainly in their expressiveness and the considered time domain and trace semantics. Temporal logics are preferably used to express statements about the future behavior of a system. The authors of [HR04] extended temporal logic with some metric construct to get quantitative temporal logic and introduced a past operator. However, the syntax of these specifications is very impractical for engineers. We therefore facilitate timing specification also through the use of natural language.

A method for clustering and specifying higher-level modes is provided by David Harel's state diagrams and superstates [Ha87]. We complement this operational view by declarative approach more concisely covering contextual information before and after a mode change. This interfaces directly with contract-based design principles, bringing mode management into their purview.

Böde at al. [Bö17, Bö19] developed natural language patterns for specifying timing behavior in contracts. They limited themselves to specifying periodically occurring events and expressing a latency between related events. These patterns were extended by Kröger et al. [Kr22, Kr23] with a mode-dependent approach. As a result, mode-dependent behavior as well as mode changes can be specified with patterns. However, the main focus of these previous works was on defining component behavior and specifying modes, leaving mode change to be handled by the component. Until now, these patterns could only be used to specify occurrences of events or latency between all occurred events. With our work, it should be possible to react only to selected events. In addition, it was not possible to specify the mode mapping between sub- and top-level components directly in the contracts.

AUTOSAR [AU22] already includes techniques that allow the integration of a mode manager. The mode manager is considered to be a separate software component and provides services for changing modes. A similar concept is to be used in the contract-based design framework. However, AUTOSAR does not provide any contract-based specification.

3 Basic Concepts

In this paper, we build on the system model from [Kr22, Kr23], see Figure 1, where we consider the system S as a component model consisting of a hierarchically nested set of components C. Each component $c \in C$ is equipped with so-called ports, on which the behavior of the component's environment as well as its own behavior becomes visible. We differentiate between the port types *event* and *variable*. Each component has a set of event ($\mathcal{P}_e$) and variable ($\mathcal{P}_v$) ports, $\mathcal{P} = \mathcal{P}_e \cup \mathcal{P}_v$. We adhere to the definitions from [Kr22, Kr23] which are described below. Only ports with identical types can be connected in the component model.

Event Ports: We partition the set of event ports $\mathcal{P}_e$ into input and output ports, i.e. $\mathcal{P}_e = \mathcal{P}_i \cup \mathcal{P}_o$. Each event port has a value domain Σ. Behaviors are observed in the form of events that occur at event ports. We use dense time and define $\mathbb{T} = \mathbb{R}_{\geq 0}$ as the time domain. An event is a tuple $e = (\sigma, t)$ consisting of a value $\sigma \in \Sigma$ and a time point $t \in \mathbb{T}$. The event port has a value at certain points in time. At all times outside of designated events, the event port has no value, which is denoted by the value $\perp$. The semantics of event ports are described using timed traces, as defined in [Bö19]. A timed trace of port p is an infinite sequence $\omega_p = (\sigma_i, t_i)_{i \in \mathbb{N}}$ where $\sigma_i \in \Sigma_p$ are elements of the value domain of port p and $(t_i)_{i \in \mathbb{N}}$ forms a monotonic sequence of time instances that is non-zeno.

Variable Ports: Variable ports $\mathcal{P}_v$ have a range of values V_p, but they have a defined value at each point in time $t \in \mathbb{T}$ which can be changed at discrete points in time. The behavior of a variable port is described by the function $v_p : \mathbb{T} \rightarrow V_p$, which assigns a value to each time point $t \in \mathbb{T}$. The variable port's value can be changed by a $set(p, v)$ event, where $v \in V_p$ and $p \in \mathcal{P}_v$. In order to respond to a change at the variable port, we can refer to a $change(p, v)$ event, which indicates a value change at $p \in \mathcal{P}_v$ to the value $v \in V_p$.

Modes: A mode represents a set of internal states of a component that are not visible to other components. Externally, modes are an abstract view to enable the differentiation of expression of different behaviors. This results in a structure that reduces the complexity of the component. Additionally, modes can serve as a safety mechanism to enable basic functionality in the case of errors. In any case, it is essential that mode changes are appropriately specified in the form of transitions and do not lead to inconsistent or incorrect behavior. In this work, the mode M of a component $c \in C$ is described by a variable port $p_{mode} \in \mathcal{P}_v$. The value of the mode port indicates the current component mode.

4 Mode Management

We now show how to integrate a mode management component into contract-based design. A mode manager has the main task according to [IS20] to initiate coordinated, safe and correct mode changes. With a mode manager, it is possible to map the modes of the lower-level components to the modes of the top-level component according to the mode specification of the mode manager. Firstly, we establish several constraints, namely that components can no longer change their own mode. In the case that a mode management component exists, all other components must send a request if they want to change their mode. After receiving the request, the mode manager will change the mode of the component that sent the request and, if necessary, of other affected components.

To realize a mode manager, we use the system model presented in [Kr22, Kr23] and make appropriate adjustments. First we assign an extra event output port $p_{req} \in \mathcal{P}_e$ to each component that can send a mode change request. It is quite conceivable that there are components that have different modes, but are completely passive and are not able to request a mode change. These components don't have a p_{req} port, but different modes. The request

port of a component has the same value domain as their mode port $p_{mode} \in \mathcal{P}_v$, which represent the current mode of this component, with $\Sigma_{p_{req}} = V_{p_{mode}}$. The target mode for a mode change is identified by the value of the request event.

Along with the addition of request ports, a new component is introduced to the system, the mode manager. Figure 2 illustrates the new component model. The mode manager has multiple variable (orange) and event ports (green) which are connected to the component variable ports that represent the component's mode. The event ports are connected to the request ports of the component to get mode requests. The mode manager responds to these requests and initiates a mode change for the affected components by changing the value of the variable ports. In addition to the variable ports used for reading and changing the component modes, the mode manager has its own mode port. The mode of the mode manager is defined by the current modes of the components, and it reflects the mode of the top-level component. For each variable port, we differentiate between whether it can be modified by a component or is read-only. We specify that the value of a variable port, which represents the mode of a component, can only be modified by the mode manager component. As the mode manager's variable ports are linked to the components' mode ports, the value is modified on the side of the component's mode port. The mode port of the mode manager is the only exception. This can be changed by the mode manager itself.

An important aspect of the mode manager is that it cannot simply reject component requests without cause. It is required to respond to every request made by a component. Consequently, components are contractually assured that their mode change requests will be carried out in a specified time. The mode manager can only delay the mode change for a maximum period of time, e.g. to change the mode of other components first, if otherwise consistency problems would occur, or in order to finish open activities. This guarantees that the mode change is not infinitely postponed or rejected, particularly in safety-critical scenarios. A problem here is the occurrence of mode requests at the same time from components which are dependent on each other. For instance, in Figure 2, Comp1 is in mode *A* while Comp2 is in mode *C*. Comp1 sends a request to change from mode *A* to *B*, which requires Comp2 to remain in mode C. However, at the same time, Comp2 requests to change from mode *C* to *D*. In this case, the mode manager has to address both requests in accordance with the guaranteed assurances.

There are different approaches to tackle that problem. The solution we assume here is a resolution function to address potential collisions between mode changes. The resolution function must specify how the mode manager should respond. On the specification side, it is necessary to include this resolution function in the VIT but how the resolution function looks like is not the scope of this paper.

A precise specification of the mode manager determines when it must change the mode of the components and which mode it takes regarding the component modes at its level of granularity. According to ISO-TR4804 [IS20], the mode manager possesses operating mode awareness, which allows it to identify the mode of each component at its level. In a

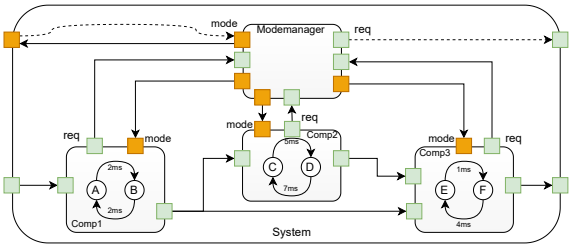


Fig. 2: Component model of the example system with mode manager

complex system, multiple mode managers exist at different levels of granularity. In this case, the mode manager acts as a request generator for higher-level components by requesting a mode change from the higher-level mode manager. After its mode is changed, it changes the modes of the components at its level of granularity. When dealing with a mode manager that is not at the highest level of granularity, it is important to take into account the dashed connection in Figure 2 between the mode port of the mode manager and the system mode port instead of the solid connection, as well as the dashed connection from the req port of the mode manager.

5 Specification Pattern

Components typically modify their mode based on their inputs, which may result from specific events, such as the activation of the component, or a particular sequence of events, such as a sequence of values or a designated number of events within a defined time interval. For instance, if inputs abruptly slow down or cease altogether, and the full functionality can not guaranteed anymore, the component should degrade. To be able to specify this behavior, we first need expressions to write conditions in the specifications based on observable. This observable are our traces which represent the behavior of the component and the environment.

Since the introduced concept from Section 4 requires components to request a mode change through request events, it is essential to define the conditions for the request. Currently, we can trigger a mode request in response to an event [Bö19, Kr22, Kr23], but this method causes requests to occur at the same frequency as the triggering event. If we consider a component's inputs as triggering events, mode requests would occur at excessively high frequencies, leading to an event overload and, therefore, wasted resources. Furthermore, we only wish to initiate a request under specific circumstances.

To begin, we establish possible conditions that must be met in order to generate a request. To define these conditions, we use MITL as described in [AFH96] because of a more powerful expressiveness. Specifically, our approach involves an infinite trace of the form $\omega = (\sigma_i, t_i)_{i \in \mathbb{N}}$ previously described in Section 3. Using $\omega(t)$, we identify the corresponding

event with value σ at time t. If the trace has no event at this point in time, we will receive the value $\perp$.

A time interval is represented as either $[a, b]$, $[a, b)$, $(a, b]$, or (a, b), with $a, b \in \mathbb{T}$. If we have an (left/right) open interval, we can use the term $l(I)$ for the lower bound and $r(I)$ for the upper bound of the interval I. Given an event trace of the form $\omega = (\sigma_i, t_i)_{i\in\mathbb{N}}$ and the interval $I = [a, b]$. We define a monotonic subsequence of this trace in the interval I by

$$\omega \mid_b^a = \left\langle (\sigma_i, t_i), (\sigma_{i+1}, t_{i+1}) \cdots (\sigma_{i+k}, t_{i+k}) \right\rangle \tag{1}$$

with $t_i \geq l(I)$ and $t_{i+k} \leq r(I)$ which contains all event tuples of the trace ω whose time t_i is within the interval I. In case of a closed interval I we get $l(I) = a$ and $r(I) = b$. We consider two special cases of this subsequence. The first is the subsequence $\omega \mid_t^0$ which describes the whole prefix at time $t \in \mathbb{T}$ of trace ω. The second, $\omega \mid_{max(t-l(I),0)}^{max(t-r(I),0)}$ identifies the partial prefix of ω between $t - r(I)$ and $t - l(I)$. With $max(t - r(I), 0)$ and $max(t - l(I), 0)$ we ensure that we only consider events that are within our time domain $\mathbb{T} = \mathbb{R}_{\geq 0}$.

We use MITL formulas to express the conditions that must be fulfilled by a trace. When using regular expressions in MITL or LTL, we always express the evolution of sequences or traces in the future. However, as we intend to employ MITL to indicate whether a condition in the trace has been fulfilled, we introduce the atomic past versions $\Diamond_I$ and $\boxminus_I$. Formally we say a trace $\omega = (\sigma_i, t_i)_{i\in\mathbb{N}}$ fulfills the MITL formula $\phi := \Diamond_I \psi$ or $\phi := \boxminus_I \psi$ at time point t if we have a subsequence $\omega \mid_{max(t-l(I),0)}^{max(t-r(I),0)}$ that fulfills ϕ:

$$\begin{aligned} \omega \models_t \Diamond_I \psi \text{ iff } \exists \omega \mid_{max(t-l(I),0)}^{max(t-r(I),0)} & \quad \text{where somewhere } \psi \text{ holds} \\ \omega \models_t \boxminus_I \psi \text{ iff } \exists \omega \mid_{max(t-l(I),0)}^{max(t-r(I),0)} & \quad \text{where } \psi \text{ holds throughout} \end{aligned} \tag{2}$$

Generation pattern: After the introduction how we express a condition for a request, we now want to show our pattern to generate an event based on a fulfilled condition. The generation pattern that generates events based on an underlying condition $Cond ::= \phi$ is given by

`Generate` *Event* `whenever` *Cond* `holds within` I`.`

The pattern is semantically defined by the language $L_{gen}(e, \phi, I^-, I^+)$, with e as the event of the generation pattern, ϕ as the condition that must be satisfied by the corresponding prefix, and the interval consisting of I^- and I^+:

$$L_{gen} = \{(\sigma_i, t_i)_{i\in\mathbb{N}} | \forall (e_i, t_i)_{i\in\mathbb{N}} : \exists t_j : \omega \mid_{max(t-l(I_c),0)}^{max(t-r(I_c),0)} \models \phi \wedge t_i - t_j \in I\}. \tag{3}$$

This means that for all generation events $(e_i, t_i)_{i\in\mathbb{N}}$ in the event trace $\omega = (\sigma_i, t_i)_{i\in\mathbb{N}}$ the following applies: it exists a t_j at which the subsequence $\omega \mid_{t_j-l(I_c)}^{t_j-r(I_c)}$ satisfies ϕ and the

occurrence of the generation event is within the interval I. We denote the interval from the condition as I_c. To make the pattern mode-based, we take the semantics from [Kr22, Kr23] and extend the generation pattern accordingly with the mode-dependent semantics. The specification pattern resulting from the extension in [Kr22, Kr23] is as follows:

`Generate` *Event* `whenever` *Cond* `holds within` I `in mode` M `.`

The extension assigns the pattern with a set of states $S = \{pre, on, post, off\}$, where it is on, off or in a transition phase (pre when turns on and post when turns off). Based on the definition that mode changes trigger events, a state trace st can be derived that contains changes from the pattern states in form of events. The projection $pr_{st,S}$ of a timed trace $(\sigma_i, t_i)_{i\in\mathbb{N}}$ over a state trace st and a set of pattern states S returns all events that occurred while the pattern was in state $s \in S$. Note that because mode changes can occur at any time, there is an infinite number of possible state traces st. We denote the set of these state traces by St. The semantics of the mode-dependent generation pattern is defined by the following language:

$$
\begin{aligned}
L^{M}_{gen} = \{(\sigma_i, t_i)_{i\in\mathbb{N}} \mid & \exists st \in St \wedge \\
& \forall (e_i, t_i) \in pr_{st,\{on,post\}}((e_i, t_i)_{i\in\mathbb{N}}) : \\
& \exists t_j : \omega \mid^{max(t-r(I_c),0)}_{max(t-l(I_c),0)} \models \phi \\
& \wedge \forall (f_i, t_i) \in \omega \mid^{max(t-r(I_c),0)}_{max(t-l(I_c),0)} : (e_i, t_i) \in pr_{st,\{on,post\}} \\
& \wedge\, t_i - t_j \in I\}.
\end{aligned}
\tag{4}
$$

Mode mapping pattern: In our approach, we obtain a mapping of the modes between sub and top-level components by mapping the values of the mode ports of the components on the mode manager side to the value of the mode port of the mode manager. To express this in a pattern, we introduce a simple mapping pattern that allows us to map the values from multiple variable ports to the value of a single variable port. The pattern is as follows:

M `whenever` *Portlist* `holds.`

M describes the value of the component mode port $p_{mode} \in \mathcal{P}_v$ to which we map the values $p_v.value$ of the other variable ports $p_v \in \mathcal{P}_v$ which are written in a list $Portlist ::= p_{v_i}.value[, Portlist]?$ with $i \in \mathbb{N}$ and $p_{mode} \notin Portlist$.

As described in Section 3, variable ports have values at all points in time. With the function $v_p : \mathbb{T} \to V_p$ we can read the values of each variable port at time t, $v_{p_v}(t) = v$ with $v \in V_p$. M_c describes the set of modes of a component c. Semantically, the pattern means:

$$
v_{p_{mode}}(t) = \begin{cases} \{M\} & \text{iff } v_{p_v}(t) = p_v.value \text{ for all } p_v \in \mathcal{P}_v \\ M_c & \text{else} \end{cases}
\tag{5}
$$

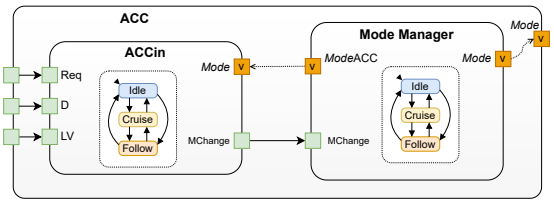


Fig. 3: Component model of the *Adaptive Cruise Control* with mode management

Tab. 1: Top-Level specification of the *ACC* with mode management

A	$\{D,LV\}$ occurs every $40\,ms$ with jitter $5\,ms$.	1
	Req occurs every $(0,\infty)\,ms$.	2
G	Reaction(Req, set($Mode$)) within $(0,5]\,ms$ in mode I.	3
	Reaction($\{D,LV\}$, set($Mode$)) within $(0,5]$ ms in mode $\{C,F\}$.	4
	Reaction(Req, set($Mode$,I)) within $(0,5]\,ms$ in mode $\{C,F\}$.	5

The semantics of a set of mode mapping patterns is the intersection of the mode sets they yield individually. The mode mapping functions are mode-independent since the mapping must remain valid across all modes and remain unchanged despite mode changes.

6 Example

To illustrate the integration of a mode manager into our component model and a possible contract specification of the mode manager, we consider an *Adaptive Cruise Control* (*ACC*) similar to the one described in [Be20, Kr22, Kr23]. We will consider a simplified version with two internal components, the *ACCin* and a *Mode Manager*. Our focus is on timing specification for mode changes and the mode manager. For this reason, further specifications have been omitted for simplicity. The component model of the *ACC* with an integrated mode manager is given in Figure 3. An *ACC* is a driving assistance system with the modes *Idle* (*I*), *Cruise* (*C*) and *Follow* (*F*). If there is a slower vehicle ahead, the *ACC* adjusts its own speed by calculating appropriate control signals and adapt its speed to the vehicle in front with an appropriate distance. If there is no slower vehicle ahead, it maintains the driver's desired speed. Initially, the *ACC* is in *Idle* mode. The *ACC* is activated by an incoming request event (*Req*) and switches to *Cruise* (*C*) or *Follow* (*F*) mode depending on the situation.

In addition to the request, the inputs of the *ACC* are the distance (*D*) and the speed of the vehicle ahead (*LV*). If the distance falls below a certain value which is indicated by the occurrence of an input event *D* with value *D.TLV* (Distance.ThresholdLimitValue), the *ACC* change to *Follow*. If the distance increases over the threshold, i.e. *D.TLV* no longer occurs, the *ACC* changes back to *Cruise*. We specify that a mode change takes up to 5ms. The top-level specification is shown in Table 1.

The *ACC* consists of two subcomponents, the *ACCin* and the *Mode Manager*. The *ACCin* has the same modes as the *ACC* - *Idle* (*I*), *Cruise* (*C*) and *Follow* (*F*) - and calculates the

Tab. 2: Specification of the component *ACCin*

A	{*D,LV*} occurs every 40 *ms* with jitter 5 *ms*.	1
	Req occurs every $(0,\infty)$ *ms*.	2
G	Reaction(*Req*,*MChange*)) within $(0,2]$ *ms* in mode *I*.	3
	Reaction(*Req*,*MChange.I*) within $(0,2]$ *ms* in mode {*C,F*}.	4
	Generate *MChange.F* whenever $\Diamond\!\!\!-_{[0,120]}$ (*D.TLV*$\rightarrow$ $\Diamond_{[0,120]}$ *D.TLV*) holds within $(0,2]$ *ms* in mode *C*.	5
	Generate *MChange.C* whenever $\boxminus_{[0,120]}$ $\neg$*D.TLV* holds within $(0,2]$ *ms* in mode *F*.	6

Tab. 3: Specification of the component *Mode Manager*

A	*MChange* occurs every $(0,\infty)$ *ms*.	1
G	Reaction(*MChange.I*,*set*(*ModeACC,Idle*)) within $(0,3]$ *ms* .	2
	Reaction(*MChange.C*,*set*(*ModeACC,Cruise*))) within $(0,3]$ *ms*.	3
	Reaction(*MChange.F*,*set*(*ModeACC,Follow*))) within $(0,3]$ *ms*.	4
	Idle whenever *ModeACC.Idle* holds.	5
	Cruise whenever *ModeACC.Cruise* holds.	6
	Follow whenever *ModeACC.Follow* holds.	7

corresponding control signals based on the inputs (*D* and *LV*). Based on the concept in Section 4, the *ACCin* cannot change its mode on its own. It has to request a mode change from the *Mode Manager*. The *Mode Manager* responds with a change of the mode of the *ACCin* by changing the value of the variable port. Change requests are requested in the form of events at the port *MChange*. When a *Req* event occurs, a mode change from or to Idle is requested. If the *ACCin* is in *Cruise* mode, a mode change to *Follow* is requested if the value *D.TLV* occurs twice within 120ms. In this case, the critical distance to the vehicle in front was violated twice, requiring a mode change to avoid a collision with the vehicle in front. If no *D.TLV* value is detected within 120ms in *Follow* mode, the vehicle changes to *Cruise*. Each request takes between 0 and 2 ms. The mode change specification of the *ACCin* is shown in Table 2.

The *Mode Manager* receives mode change requests from the *ACCin* via the *MChange* port in the form of events. By changing the associated variable port *ModeACC*, which is connected to the mode port of the *ACCin*, it sets the *ACC* to desired mode. The *Mode Manager* needs up to 3 ms to change the variable port. The maximum time for the *ACCin* to send a request is 2 ms. This results in a maximum time for mode changes in 5 ms, as specified in the top-level specification from Table 1. In addition to set the mode, the *Mode Manager* provides a unique mapping of modes between lower-level and higher-level components, in this case between the modes of the *ACCin* and the *ACC*. In this example, the mapping is one-to-one, so that the modes of the *ACCin* correspond to the modes of the *ACC*. The *Mode Manager* specification is shown in Table 3. Figure 4 shows example traces of the behavior of the specified *ACC*. The *ACC* is immediately started by a *Req* event. The *ACCin* requests a change to *Cruise* by a *MChange.C*. After receiving the *Mode Manager* change the *ACCin* into *Cruise* mode. After a *D.TLV* occurred two times in 120 ms, the *ACCin* requests a change from *Cruise* to *Follow* mode by a corresponding *MChange.F* event, see $T = 124$ ms. After the change to *Follow*, an *MChange* event to change to *Cruise* is generated after 120 ms at the earliest. This

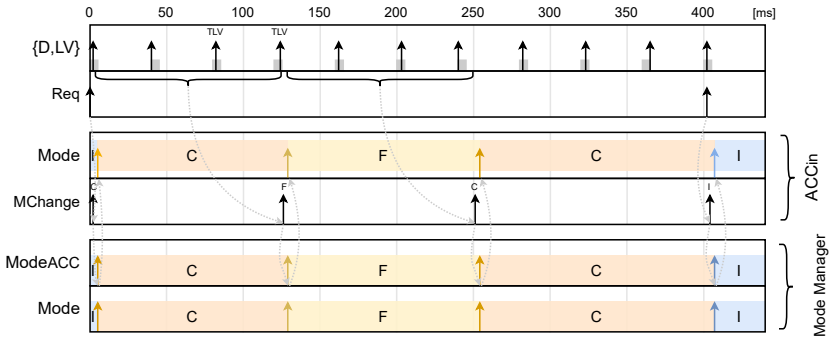


Fig. 4: Example trace of the *Adaptive Cruise Control* with mode management

is at time $T = 249$. At time $T = 400$ ms the *ACC* is changed back to *Idle* by a *Req* event. The resulting dependencies and sequences are shown by arrows.

7 Conclusion

The presented approach reduces the specification complexity for multimodal systems by introducing a central mode management component in contract-based design. Mode changes are triggered by the respective components in the form of events triggered by conditions specified in temporal logic. The events are received and handled by the mode management component. The contract specification of the mode manager also provides a mapping between the sub-level and top-level components. Our specifications are mostly based on natural language patterns. Only the conditions triggering a mode change are expressed in MITL for reasons of expressiveness.

This work has been funded by the *Federal Ministry of Education and Research* (BMBF) as part of *AutoDevSafeOps* (01IS22087Q).

Bibliography

[AFH96] Alur, Rajeev; Feder, Tomás; Henzinger, Thomas A: The benefits of relaxing punctuality. Journal of the ACM (JACM), 43(1):116–146, 1996.

[AU22] AUTOSAR: , Guide to Mode Management, 2022.

[Be08] Benveniste, Albert; Caillaud, Benoît; Ferrari, Alberto; Mangeruca, Leonardo; Passerone, Roberto; Sofronis, Christos: Multiple Viewpoint Contract-Based Specification and Design. In (de Boer, Frank S.; Bonsangue, Marcello M.; Graf, Susanne; de Roever, Willem-Paul, eds): Formal Methods for Components and Objects. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–225, 2008.

[Be18] Benveniste, Albert; Caillaud, Benoît; Nickovic, Dejan; Passerone, Roberto; Raclet, Jean-Baptiste; Reinkemeier, Philipp; Sangiovanni-Vincentelli, Alberto; Damm, Werner; Henzinger, Thomas A.; Larsen, Kim G.: Contracts for System Design. Foundations and Trends in Electronic Design Automation, 12(2-3):124–400, 2018.

[Be20] Bebawy, Yosab; Guissouma, Houssem; Vander Maelen, Sebastian; Kröger, Janis; Hake, Georg; Stierand, Ingo; Fränzle, Martin; Sax, Eric; Hahn, Axel: Incremental Contract-Based Verification of Software Updates for Safety-Critical Cyber-Physical Systems. In: 2020 International Conference on Computational Science and Computational Intelligence (CSCI). IEEE, 2020.

[Bö17] Böde, Eckard; Büker, Matthias; Damm, Werner; Ehmen, Günter; Fränzle, Martin; Gerwinn, Sebastian; Goodfellow, Thomas; Grüttner, Kim; Josko, Bernhard; Koopmann, Björn; Peikenkamp, Thomas; Poppen, Frank; Reinkemeier, Philipp; Siegel, Michael; Stierand, Ingo: Design Paradigms for Multi-Layer Time Coherency in ADAS and Automated Driving (MULTIC). 302. Research Association for Automotive Technology, 2017.

[Bö19] Böde, Eckard; Damm, Werner; Ehmen, Günter; Fränzle, Martin; Grüttner, Kim; Ittershagen, Philipp; Josko, Bernhard; Koopmann, Björn; Poppen, Frank; Siegel, Michael; Stierand, Ingo: MULTIC-Tooling. 316. Research Association for Automotive Technology, 2019.

[Ha87] Harel, David: Statecharts: A visual formalism for complex systems. Science of computer programming, 8(3):231–274, 1987.

[HR04] Hirshfeld, Yoram; Rabinovich, Alexander: Logics for real time: Decidability and complexity. Fundamenta Informaticae, 62(1):1–28, 2004.

[IS20] ISO Central Secretary: Road vehicles - Safety and cybersecurity for automated driving systems - Design, verification and validation. Standard ISO/TR 4804:2020(E), International Organization for Standardization, Geneva, CH, 2020.

[Ka15] Kaiser, Bernhard; Weber, Raphael; Oertel, Markus; Böde, Eckard; Nejad, Behrang Monajemi; Zander, Justyna: Contract-Based Design of Embedded Systems Integrating Nominal Behavior and Safety. Complex Systems Informatics and Modeling Quarterly, (4):66–91, 2015.

[Ko90] Koymans, Ron: Specifying Real-Time Properties with Metric Temporal Logic. Real-Time Systems, 2(4):255–299, 1990.

[Kr22] Kröger, Janis; Koopmann, Björn; Stierand, Ingo; Tabassam, Nadra; Fränzle, Martin: Handling of Operating Modes in Contract-Based Timing Specifications. In (Nouri, Ayoub; Wu, Weimin; Barkaoui, Kamel; Li, ZhiWu, eds): Verification and Evaluation of Computer and Communication Systems. volume 13187 of Lecture Notes in Computer Science. Springer, Cham, pp. 59–74, 2022.

[Kr23] Kröger, Janis; Koopmann, Björn; Stierand, Ingo; Fränzle, Martin: Contract-based specification of mode-dependent timing behavior. Innovations in Systems and Software Engineering, pp. 1–17, 2023.

[Pn77] Pnueli, Amir: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science. IEEE, pp. 46–57, 1977.

[SVDP12] Sangiovanni-Vincentelli, Alberto; Damm, Werner; Passerone, Roberto: Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*. European Journal of Control, 18(3):217–238, 2012.

6th Workshop on Avionics Systems and Software Engineering (AvioSE'24)

A Universal Configuration Format for Avionics

Philipp Chrysalidis [1], Frank Thielecke[1]

Abstract: Avionics module configuration, especially in the face of advancing technologies, will become more complex as computational demands rise. This requires a robust and automated approach while adhering to industry standards. However, state-of-the-art configuration is still highly error-prone and suffers from various stakeholders working with unsynchronized and decentralized data. This causes unnecessary iterations, leading to delays in development. The Universal Configuration Format for Avionics (UCoF), integrated into the AvioNET framework, presents a forward-looking solution. UCoF, built upon a meta-model approach, strives to enhance the configuration process through model-based methods. It meets essential configuration management requirements and offers versatility by supporting the configuration of diverse avionic platforms. Combining essential data for configuring real avionics device families, implementation targets and network design grants users access to a comprehensive data set throughout the configuration process.

Keywords: Avionics; Configuration; MBSE

1 Introduction

The configuration of avionics modules is a complex and still highly error-prone process, especially with new technologies on the horizon (e.g., single-pilot cockpit, smart cabin) that demand higher computational capabilities. Moreover, a more robust and automated configuration process is essential for efficiently implementing these new technologies.
However, aviation standards provide a well-defined framework for development, including the configuration of Integrated Modular Avionics (IMA) modules as defined in ARINC 653 [Ae12]. Since IMA is integral to aviation, any configuration process must adhere to these standards.
Furthermore, module and application development and testing are distributed and often collaborative processes. These trends are likely to intensify, with advances in model-based systems engineering (MBSE) leading to more virtual and hybrid testing. To ensure smooth operations and minimize delays in development under these circumstances, transparency and consistency of information are of utmost importance. Without these, faults may propagate to later development phases, leading to time-consuming iteration loops. Meanwhile, there is currently no common industry standard for avionics configuration, and stakeholders often rely on proprietary formats.
To address this challenge, the Institute of Aircraft Systems Engineering (FST) at the Hamburg

[1] Hamburg University of Technology, Institute of Aircraft Systems Engineering, Nesspriel 5, 21129 Hamburg, Germany, firstname.lastname@tuhh.de

doi:10.18420/sw2024-ws_04

University of Technology (TUHH) has developed the Avionics Next-Gen Engineering Tools (AvioNET) framework [HT21]. AvioNET is a seamless end-to-end toolchain based on generic MBSE methods, offering solutions for a more efficient design process throughout all development phases. Key elements of AvioNET include Architecture, Configuration, Simulation, Testing, Verification & Validation, Visualization, and Avionics Data Management. Developing methodologies within this framework ensures immediate consideration of all necessary interfaces for seamless information transfer during development. Utilizing the AvioNET framework, an approach for automating the configuration process, has already been implemented at FST. The model-based method outlined in [CHT23] automatically derives the configuration for an avionics module from a set of requirements. However, AvioNET's scope extends to platform solutions, necessitating a more generic approach that includes data management for data consistency.
The Universal Configuration Format for Avionics (UCoF), introduced in this paper, is integrated into AvioNET. This approach enables model-based configuration not only for avionics hardware but also for virtual and testing environments, facilitating Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) testing for continuous integration and validation.
This paper will contextualize current research in section 2. Section 3 will lay out the requirements regarding configuration management. Section 4 will present the UCoF approach. Finally, the paper will be summarized in section 5, with an outlook for future research provided.

2 Related Work

In 2010, Horváth et al. [HVS10] presented a framework for systematically designing ARINC 653 configuration tables for the Wind River VxWorks Real-Time Operating System (RTOS). Using meta-modeling as proposed by the ARINC 653 standard, they created a data model from which specific design instances could be derived. To align the meta-model with IMA development roles, the model was divided into four subgroups: Applications, Health Monitoring, Communication, and Interface Control Document. This concept was implemented in the Eclipse Modeling Framework (EMF) [Fo23], enabling the use of the model in their overall design approach and toolchain. However, this approach heavily focused on ARINC 653 implementation and lacked universal applicability.
Darif et al. [Da22] took a similar approach to embed a model-based approach into a tooling environment for an RTOS. Their goal was to support multiple RTOS to reduce certification costs by introducing a high degree of reusability with their concept. Additionally, their tooling could generate certification data for ARINC, such as configuration tables and test data.
In contrast to the previous works, Annighoefer [An19] employed a different approach to configuration. Instead of replicating the ARINC 653 standard in a meta-model, they used a generic avionics architecting model. This model aimed to perform on all levels of detail during the design of avionics components, allowing for a wide range of modeling targets.

Therefore, the model could be used for configuring proper ARINC 653 certified hardware [An20].
Furthermore, a query language for meta-models expanded the generic capabilities and enabled easy access to the model. By combining both the query language and the generic avionics model, interfaces for data transfer from and to other formats could be easily created. These methods facilitated highly automated processing of configuration data in tooling environments, as seen in Mueller et al. [Mu23]. However, it's worth noting that the very generic approach has a significant disadvantage, since participants could interpret the model differently, leading to compatibility issues.
Halle; Thielecke [HT09] presented the challenges faced during the configuration process due to the involvement of various stakeholders while needing to maintain data consistency. They introduced a meta-model approach, combining it with data management, to support validation, continuous integration, extensibility of the format, and interfaces to other formats or tools. UCoF builds strongly upon these works, further expanding on the presented solutions and broadening the defined scope.

3 Model-Based Avionics Configuration Management

Considering more recent developments described by Martinen et al. [Ma17] and Uludağ et al. [Ul23], information flow between stakeholders will become even more complex than what was described in [HT09]. Test rigs will gain even more prominence in development and may be locally distributed, communicating through long-range internet connections. Examples of these concepts can be found in the works of Chrysalidis et al. [CHT22] and Martinez et al. [MGG22]. In the latter case, the newly created standard by EUROCAE [EU20] for distributed testing was successfully utilized, indicating the path toward future developments. Information must be readily available across different locations with a guarantee of uncompromised data.
The advances in virtual integration also allow for more reliable Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) testing earlier in the process. However, the setup of the required virtual or hybrid environments introduces an added layer of configuration complexity. Increasing the efficiency of the configuration process, therefore, includes solutions for these environments. The scope of UCoF encompasses the formalization and standardization of the configuration of testing environments while still needing to guarantee uncompromised data between all participating stakeholders.
Moreover, as standards evolve through updates and expansions, and new standards continue to emerge, UCoF's sustainability relies on its ability to readily accommodate these evolving specifications.

3.1 Requirements for Configuration Management

Based on the outlined scope, we present the requirements for a configuration management format capable of meeting future needs in Table 1.

Requirement	Description
Satisfy Standards	The configuration format has to satisfy the most common and most important standards and process requirements in avionics development.
Modularity	Modularity is needed, so that different parts of the configuration can be completed independently. It is also a strong basis for the reusability requirement.
Reusability	An efficient process greatly profits from a high degree of reusability, ensuring less redundant tasks.
Traceability	Changes to the configuration need to be traced to both a time and a stakeholder. This way, faults can be identified more clearly.
Accessibility	Accessibility means both easy access to the model (i.e. open source) and intuitive use. This means, that the usage domain must be clearly defined.
Generality	The model must be as generic as possible, as to incorporate the maximum amount of information for the maximum amount of systems, with the least amount of redundant data.
Automatibility	The format must be machine-readable and accessible through parsing.
Adaptability	The configuration must be open to new technologies, while still maintaining backwards compatibility, as to ensure smooth integration of new configuration targets.
Information Propagation	Information must flow bidirectionally throughout the configuration process, meaning that both a source and a sink for information must be provided, including corresponding interfaces.
Scalability	Configurations need to be stored efficiently and accessible for all project sizes.

Tab. 1: Requirements for Configuration Management

4 Universal Configuration Format for Avionics

UCoF is built upon the requirements outlined in subsection 3.1 and aims to be a future-proof solution for avionics configuration. UCoF is based on a meta-model which is implemented using EMF [Fo23]. The process of creating a graphical user interface is streamlined through the automation of code generation, allowing for the rapid assessment of the meta-model's effectiveness in specific instances.

The primary objective of UCoF is to expand the platform definition and incorporate test systems in both hardware and software, providing swift and easy access to SIL and HIL testing. By integrating these test systems as target platforms within the model, data consistency is assured, with no information concealed within transformation scripts (see Figure 1). This approach to transparency makes working with different hardware intuitive and also allows for direct information links, aiding in the debugging processes.

Moreover, changes made to target configurations can be easily traced, enabling the reuse of these configurations for different platform components. Thus, UCoF supports continuous integration of avionics configurations throughout all stages of development.

Configuration data in UCoF is categorized into three primary groups: "Devices", "Testing" and "Network" (see Figure 2). The "Devices" group encompasses all configurations for individual modules included in the platform. Module configuration is generic and independent of the target, while implementation-specific configurations for real hardware, test benches, and virtual environments are separated. Real hardware configuration is

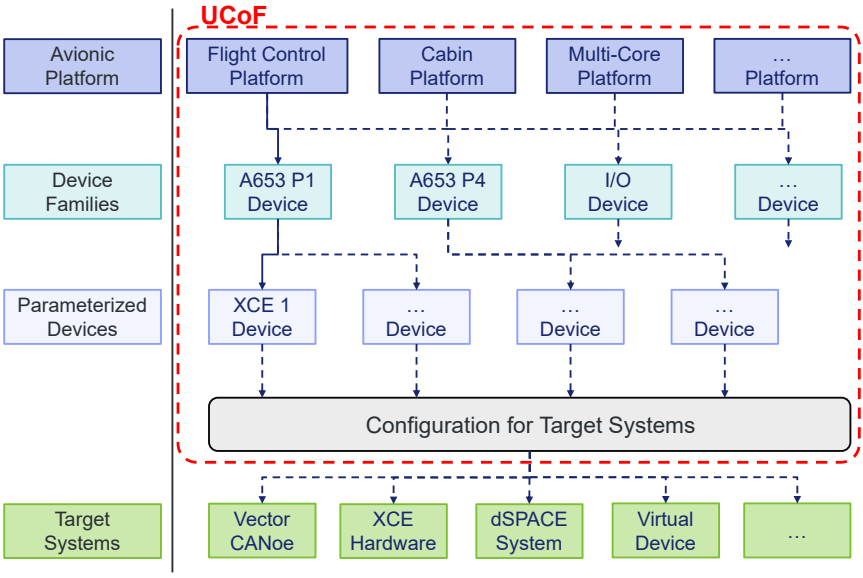


Fig. 1: UCoF Approach

also included in the “Device” group, while configurations for test benches and virtual environments are found in the “Testing” group. The “Testing” group additionally includes the definition of basic testing procedures. This allows for test bench configurations to be included in the model through references, while maintaining a clear separation from the proper avionics hardware configuration. Additionally, the definition of basic testing procedures facilitates quick test case execution by providing a central information hub through UCoF. Within the “Network"group, configurations for device communication, such as switches and gateways, are specified. The generic configuration of gateways presents an innovative strategy for tackling the challenges associated with test virtualization, as outlined in section 3.

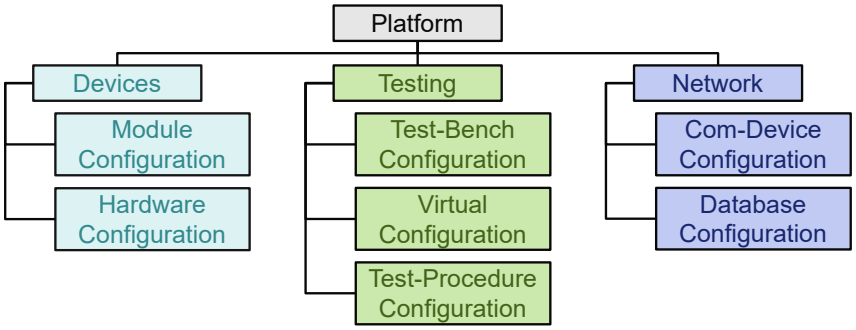


Fig. 2: UCoF Structure

4.1 Data Categorization in UCoF

Figure 3 demonstrates the categorization of data within UCoF through a simplified example. As previously explained, data is categorized into master and target specific information. Both datasets are stored separately but interconnected via references. Consequently, target-specific configurations solely extend the provided master information, allowing for independent addition of new targets. Most of the information is reused across all targets, minimizing redundancy even when accommodating various implementations. Modifications to target-specific data are isolated, while adjustments to master data resulting from tests can be proposed and accepted by multiple stakeholders.
In this example, CAN bus data is defined for a test system and an ARINC 653-compliant avionics device. General data comprises attributes like the identifier (CAN ID), data length code (DLC), and baud rate, which remain constant across all implementations and are shared among specific configurations. Target-specific configurations might involve assigning channel sets and in-device termination for test systems or drivers and hardware ports for the ARINC 653 device.
Alterations to the ARINC 653 driver or test system termination are isolated to their targets and don't affect the master configuration. However, should a too high baud rate be detected during HIL testing with the test system, this change can be automatically applied to the master data and verified. UCoF's integrated data connectivity facilitates subsequent reconfiguration of the ARINC 653 device without necessitating additional user input, since the updated data is referenced within the model. Consequently, this workflow significantly reduces the likelihood of errors and ensures consistent behavior across different targets.

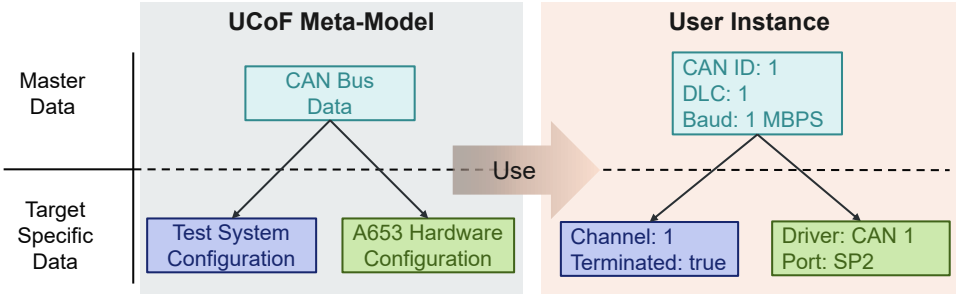


Fig. 3: UCoF Data Categorization

The implementation of this approach in UCoF is demonstrated in Figure 4 in a simplified excerpt of the meta-model. The UML classes are categorized into two main groups: master and target-specific data, further segmented into "Devices", "Network", and "Testing". Focused on CAN communication, relevant classes in the model are consolidated as "classes" to improve clarity
The modeled master device represents an ARINC 653 compliant module. In these, sampling ports can be defined for communication with other modules. Configuring a sampling port entails defining the respective communication protocol and payload. In UCoF, this

data is stored separately in a database linked to the device, enabling a clear overview of communication and facilitating data reusability.
However, to meet implementation requirements, information within the “Devices” and “Network” sections needs expansion. To achieve this, the master device configurations reference potential implementations. For instance, in the context of a test system, the CAN configuration undergoes further specification through additional attributes. The generation of the executable target configuration can only be automated with this specific, target-related information.

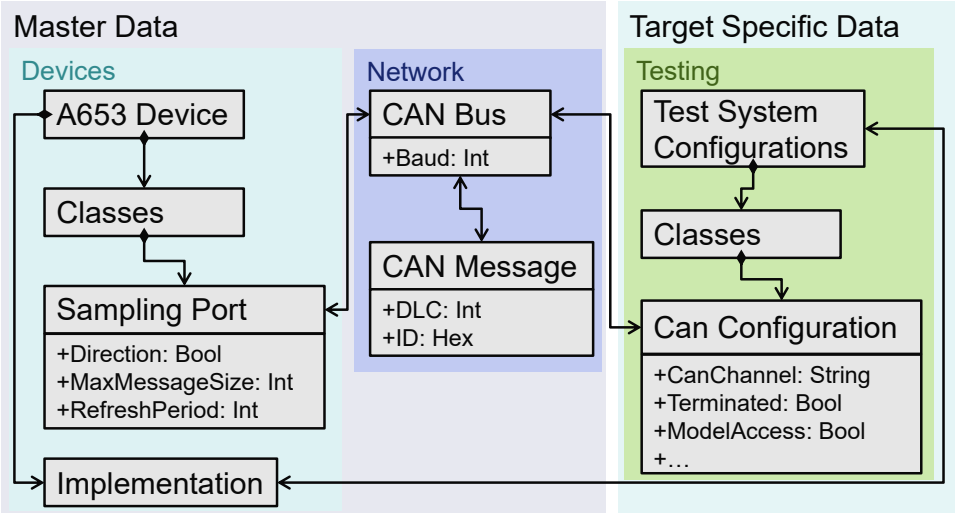


Fig. 4: Simplified UCoF UML Excerpt

Additionally, Figure 4 illustrates the benefits of UCoF’s modular approach. By linking information through references, new configuration data can be seamlessly incorporated while preserving the fundamental structure of the core elements. For instance, the configuration of an ARINC 653 compliant device maintains an adherence to the established standard, ensuring minimal structural alterations over time.
Conversely, configuring test systems proves more dynamic due to frequent updates within proprietary configuration environments. This volatility necessitates a more flexible approach to accommodate changes effectively. Moreover, the format’s capability to easily integrate new, emerging implementation targets, underlines the required future-proof architecture of the meta-model.

4.2 Working with UCoF

UCoF seamlessly integrates configuration information throughout the development cycle, supporting not only individual devices, but also facilitating configuration for the entire

avionics platform. To start, input for platform configuration, sourced either from a singular or multiple sources (such as a platform architecture), is required.
This input is further specified by relevant stakeholders, for example the system integrator defined in ARINC 653. As depicted in Figure 5, the master configuration data is derived from this input and serves as the baseline for all target configurations.
Depending on the development stage, single devices or the already defined platform can be configured for the respective targets. Early in development, when hardware might not yet be available, the master configuration can be expanded to encompass configuration information for a simulation environment. This enables a SIL approach early in the development, adding further iteration cycles for the configuration data, without any redundant information. Updates stemming from SIL testing can seamlessly be integrated into the model via interfaces that enable information flow to the UCoF model.
Advancing through the development stages, the SIL approach may transition to HIL testing, where available hardware and test systems come into play. Target specific configurations for real hardware or test systems expand the master configuration data, greatly reducing the configuration workload during this stage. The creation of executable code or other configuration artifacts, such as proprietary project files for test systems, are created through automation interfaces. These interfaces are set up bidirectional, allowing for changes to be propagated back to the UCoF model. Therefore, the platform configuration can be adapted dynamically with a minimal risk for errors. In instances where the platform remains partially virtualized, the inclusion of gateways becomes essential to enable comprehensive communication among all modules. With UCoF, the configurations of said gateways are included in the same model and are part of the overall platform configuration, thereby ensuring the holistic platform approach for all modules.
Upon the availability of actual hardware for the complete platform, expanding the master configuration data simply involves specifying configurations for the finalized modules. With automatic data updates and transfer, information continuity is preserved between testing phases and the final implementation. The model's traceability attributes ensure comprehensive tracking of changes, guaranteeing data integrity throughout the process.

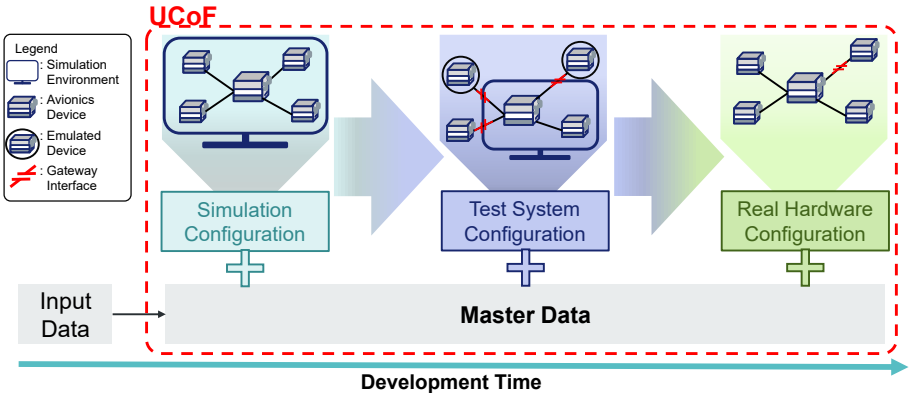


Fig. 5: UCoF during Development

5 Conlusion and Outlook

Avionics module configuration is a complex process, exacerbated by higher demand in computational power due to emerging technologies. This paper introduces the Universal Configuration Format for Avionics (UCoF) within the AvioNET framework, aiming to provide a future-proof solution. UCoF is based on a meta-model and enhances platform configuration by unifying device, network and testing configurations within a single format, ensuring data consistency and transparency through all development steps.
UCoF's design adheres to key configuration management requirements such as adherence to standards, modularity, reusability, traceability, accessibility, generality, automatibility, adaptability, and bidirectional information flow. Moreover, UCoF's segregation of master and target-specific information ensures interconnectedness while offering a highly adaptive model, enabling continuous integration. This categorization minimizes redundancy, enhancing efficiency and reliability across avionics development.
Future research will expand UCoF to encompass additional implementation targets. It will also delve into developing generic automation methods to simplify the creation of interfaces for input and output data. Furthermore, UCoF's open-source project release is planned, once a well-defined beta version is available, fostering collaborative development and broader adoption.

Acknowledgement

This work was funded by the German Federal Ministry of Economic Affairs and Climate Action (BMWK) within the TALIA project. Their support and the cooperation of the partners is greatly appreciated. The authors of this publication are responsible for its content.

Supported by:

on the basis of a decision
by the German Bundestag

References

[Ae12] Aeronautical Radio, Inc.: Avionics Application Software Standard Interface: ARINC Specification 653P1-4, Standard, 2012.

[An19] Annighoefer, B.: An Open Source Domain-Specific Avionics System Architecture Model for the Design Phase and Self-Organizing Avionics. SAE Int. J. Adv. &. Curr. Prac. in Mobility 1/2, pp. 429–446, 2019.

[An20] Annighoefer, B.; Brunner, M.; Schoepf, J.; Luettig, B.; Merckling, M.; Mueller, P.: Holistic IMA Platform Configuration using Webtechnologies and a Domain-specific Model Query Language. In: 2020 AIAA/IEEE 39th Digital Avionics Systems Conference. Virtual Conference, pp. 1–10, 2020.

[CHT22] Chrysalidis, P.; Halle, M.; Thielecke, F.: Testing of High-Lift System Functions with a Distributed Avionics Test Bench. In: 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC). Portsmouth, VA, USA, pp. 1–8, 2022.

[CHT23] Chrysalidis, P.; Hoeber, H.; Thielecke, F.: A semi-automated approach for requirement-based early validation of flight control platforms. CEAS Aeronaut. J. 14/1, pp. 271–280, 2023, ISSN: 1869-5590.

[Da22] Darif, I.; Politowski, C.; El-Boussaidi, G.; Kpodjedo, S.: A Domain Specific Language for the ARINC 653 Specification. In: IEEE International Symposium on Software Reliability Engineering Workshops. Charlotte, NC, USA, pp. 238–245, 2022.

[EU20] EUROCAE: Technical Standard of Virtual Interoperable Simulation for Tests of Aircraft Systems in Virtual or Hybrid Bench, EUROCAE ED-247A, Standard, 2020.

[Fo23] Foundation, E.: Eclipse Modeling Framework, [Online; accessed 5. Oct. 2023], 2023, URL: https://eclipse.dev/modeling/emf.

[HT09] Halle, M.; Thielecke, F.: Next Generation Konfigurationsmanagement for IMA, FlyING 2009, Oct. 2009.

[HT21] Halle, M.; Thielecke, F.: Avionics Engineering Tool Network (AvioNET):Experiences With Highly Automised and Digital Processes for Avionics Platform Development. In: 2021 AIAA/IEEE 40th Digital Avionics Systems Conference (DASC. San Antonio, TX, USA, 2021.

[HVS10] Horváth, A.; Varró, D.; Schoofs, T.: Model-Driven Development of ARINC 653 Configuration Tables. In: 29th Digital Avionics Systems Conference. Salt Lake City, UT, USA, 6.E.3-1-6.E.3–15, 2010.

[Ma17] Martinen, D. H.; Lagalaye, M.; Pfefferkorn, J.; Casteres, J.: Modular and Open Test Bench Architecture for Distributed Testing, [Online; accessed 4. Oct. 2023], 2017.

[MGG22] Martinez, R.; Gaurel, C.; Gruber, M.: Distributed Testing using VISTAS (ED-247 RevA). In: European Test and Telemetry Conference. Nuremberg, Germany, pp. 227–231, 2022.

[Mu23] Mueller, P.; Merckling, M.; Tietz, V.; Frey, C.; Luettig, B.; Waldvogel, A.; Annighoefer, B.; Abdo, K.; Halle, M.; Thielecke, F.: Introduction of a Dedicated Platform Level for IMA Systems Development With an Extensive Automation Tool Support. In: 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC). Barcelona, Spain, 2023.

[Ul23] Uludağ, Y.; Bayoğlu, Ö.; Candan, B.; Yılmaze, H.: Model-Based IMA Platform Development and Certification Ecosystem. In: 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC). Barcelona, Spain, 2023.

Enhancing DO-178C/DO-331 Based Process-Oriented Build Tool: Integration of System Composer and Automated PIL Simulation

Purav Panchal[1], Konstantin Dmitriev[2], Stephan Myschik[3]

Abstract: The growing utilization of software in safety-critical systems can be attributed to advancing technology and substantial interest within aerospace and space industries. However, this increased reliance on software to enhance avionic system functionality raises crucial safety questions, emphasizing the need for compliance with standards like DO-178C/DO-331. To facilitate development, a process-oriented build tool was created in MATLAB/Simulink. This tool enhances development efficiency and ensures adherence to established processes, offering benefits like modular software management, systematic artifact handling with traceability, seamless integration with various verification tools, automated model and code verification, and a well-defined design environment. Recently, two new advancements have been made to the tool, integration of System Composer for developing software architecture and automated processor-in-the-loop (PIL) verification using Trace32. This paper presents these new developments along with examples.

Keywords: DO-178C; DO-331; model-based software; process-oriented; software architecture; verification

1 Introduction

Systems whose malfunction can result in human harm, damage to individuals, or environmental harm are classified as safety-critical systems. Instances of such systems span multiple domains, including nuclear, medical, aerospace, military, automotive, and space. The software within these systems necessitates extra safety considerations during development and must adhere to specific standards. In the case of aerospace, safety-critical software development must comply with the DO-178C process standard.

DO-178C document is an acceptable means of compliance to the regulations for developing airborne software [DC11]. DO-331, a supplement to DO-178C, provides guidance for model-based software development and verification [DO11]. These documents specify numerous required objectives, activities, and artifacts for each software life cycle phase,

[1] University of the Bundeswehr Munich, Institute for Aeronautical Engineering, Werner-Heisenberg-Weg 39, 85579 Neubiberg, Germany purav.panchal@unibw.de
[2] Technical University Munich, Institute of Flight System Dynamics, Boltzmannstraße 15, 85748 Garching, Germany konstantin.dmitriev@tum.de
[3] University of the Bundeswehr Munich, Institute for Aeronautical Engineering, Werner-Heisenberg-Weg 39, 85579 Neubiberg, Germany stephan.myschik@unibw.de

doi:10.18420/sw2024-ws_05

from planning to certificate authorities. The number of objectives to meet depends on the software's criticality level, indicated by the Design Assurance Level (DAL). However, these process standards demand rigorous methodologies, extensive documentation, strong artifact traceability, and various verification activities. Implementing such resource-intensive processes poses a challenge, particularly for small-scale industries. These methodologies often diverge from agile practices, and changing requirements in later development stages can lead to substantial cost and effort increases, known as the 'big-freeze' problem [Cl21].

To address these limitations, a process-oriented build tool Mrails has been developed and employed in various safety-critical applications, such as flight controllers, battery controllers, and motor controllers at the Institute for Aeronautical Engineering, University of the Bundeswehr Munich, and the Institute of Flight System Dynamics, Technical University of Munich. The tool's application has significantly improved process conformance, streamlined the development of complex modular software, and simplified artifact management and traceability. Built using MATLAB/Simulink, this tool offers a development framework via a user-friendly MATLAB command line interface.

The build tool Mrails is continuously being applied for several applications and is also being improved in parallel. [Pa22b], [HMH19], [Pa23a], [Pa23c], [Pa22c] emphasizes on the several applications like safety-critical flight controller and battery controller developed using the build tool. [Pa22a] explains several modifications like the improvement in code generation for complex software structure. An overview of this build tool is presented in [Pa23b, Dm20] which explains the workflow along with all the features. In this paper, three new automated jobs have been added to the tool: (i) to create skeleton software architecture model in System Composer, (ii) to deploy the developed software architecture and (iii) to execute requirements-based test cases in processor-in-the-loop (PIL) mode using Trace32 debuggers [La23]. The remainder of the paper is structured as follows: Section 2 explains the complete software development process along DO-178C from requirements to software verification and also the traceability concept. This section also explains the tool Mrails briefly. Section 3 states the two aforementioned improvements of the build tool and finally, section 4 concludes the paper and discusses future work.

2 DO-178C Based Software Development Toolchain

This section will present an overview of the complete software development toolchain with the required tools. Implementing the widely recognized V-Model in software development enhances software safety by enabling verification and validation at different developmental stages. This approach aids in early defect detection and aligns with DO-178C/DO-331 standards. When combined with model-based software development, the V-Model not only maximizes the benefits of a systematic approach but also provides the flexibility to integrate models as requirements, visualize the software, facilitate automated code generation and verification, and enhance collaboration within distributed development teams.

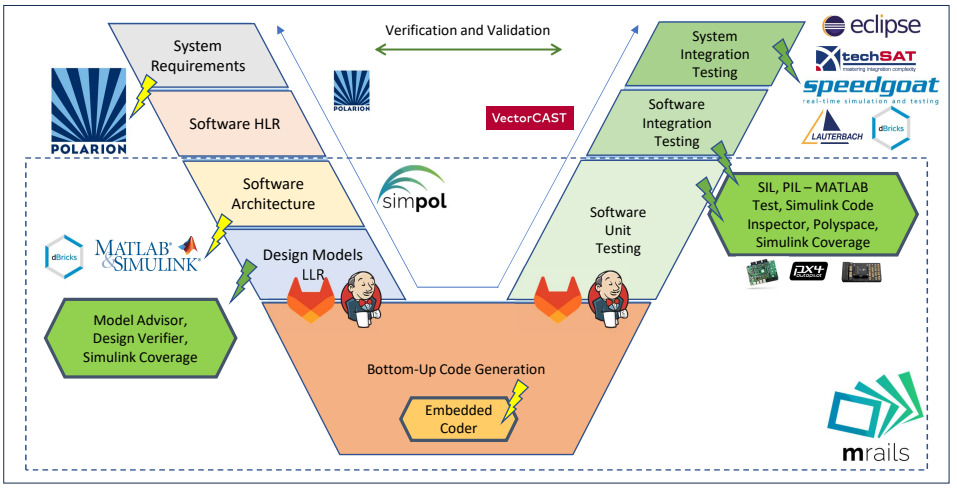


Fig. 1: DO-178C/DO-331 based software development toolchain showcasing development phases with the associated tools

The software is divided into two components: functional code (incorporating logic) and an application framework (comprising interfaces and low-level driver code). The model-based software development V-Model for the functional part is illustrated in Figure 1. The toolchain is presented here to create a base for the research, for example, the improvements like the addition of the software architecture is now a part of the build tool process. The toolchain also opens several opportunities of automation in each stages of development and testing.

The process commences by defining stakeholder or customer requirements, which then shape the project's objectives and system requirements. Siemens Polarion PLM [Si04] is utilized to document system requirements, software and hardware requirements, test cases, software and hardware development plans, and change control and problem reporting.

Subsequently, the hardware development follows a similar process as software, but it's beyond the scope of this research. Software high-level requirements derived from the system requirements are then linked to the software architecture model designed using MathWorks System Composer. Software low-level requirements are in the form of design models developed in MATLAB/Simulink. The development of design models utilizes the process-oriented build tool Mrails, which offers a modular model-based development framework in MATLAB/Simulink. It automates jobs using Embedded Coder to generate code for the functional part in MATLAB, aligning with DO-331 guidelines and performing static model analysis.

For the application software, low-level software requirements are stored in textual form in Polarion, while the code for the application software part is manually written in Eclipse IDE [NX23]. After code generation, static testing is conducted using Polyspace, and simulation

testing is performed using Simulink Tests and VectorCAST [Ve23] for the functional and application parts, respectively. The functional software is then integrated into the application framework in the Eclipse project and tested in real-time using a hardware-in-the-loop (HIL) testbench. The interfaces between software and hardware are managed in dBricks [dB23], an interface management tool. Various tools ensure traceability from requirements to testing.

To maintain an agile workflow, Git is used for version control, and a structured workflow is followed. A Jenkins based continuous integration server is set up for automated testing and closed-loop requirements verification.

2.1 Process-Oriented Build Tool

The improvements presented in this paper are part of the build tool and hence it is necessary to understand the background of the tool along with its features. The build tool not only optimizes the design and verification tasks but also offers process-oriented features throughout the software development cycle like design, test and traceability review checklists derived from the DO-178C/DO-331 standards. Figure 2 shows the steps that are applied when using the build tool.

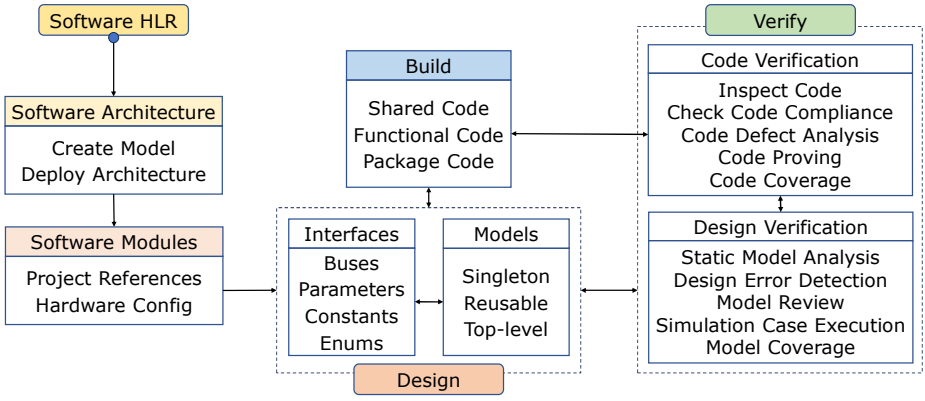


Fig. 2: Workflow of the process-oriented build tool along with the automated tasks provided by the tool

Some of the key features are discussed below:

1. Defined Development Environment and Modular Software Structure: The tool establishes a development environment encompassing tool configuration, linking, version consistency, modeling guidelines, coding guidelines, and standard naming conventions. This fosters a modular software development process within distributed teams, mitigating configuration mismatches and saving time and effort in software component integration. The tool seamlessly manages dependencies and integrates multiple

software components without incurring configuration mismatches or regenerating artifacts.

2. Model Scaffolding: The build tool offers a set of pre-configured “creators"that enable developers to easily create models, data dictionary items, such as bus elements and constants, and Simulink tests. These actions occur without the need to worry about configuration settings and storage directories.

3. Incremental Code Generation: Unlike the top-down approach of conventional code generation processes (such as MathWorks Embedded Coder), where code for interfaces like Simulink buses is rebuilt upon changes in other components, the build tool follows a staged bottom-up code generation approach. This method generates code at the module level before integrating it into the top level.

4. Automated Design and Code Verification and Testing Tasks: The build tool’s life cycle package includes a list of automated design and code verification jobs, with results accumulating in an incrementally generated status report.

5. Process-Oriented Review: In addition to automated tasks, the life cycle package provides manual checklists for model, traceability, and simulation reviews. The build tool offers a straightforward infrastructure to add these checklists in accordance with established standards.

6. Artifacts Traceability: The tool ensures comprehensive traceability between artifacts. For example, verification task results are stored in a web-based interface that provides up-trace and down-trace links to trace artifacts like models, jobs, data dictionaries, and more related to a specific task. Beyond verification tasks, the tool maintains detailed traceability of build information after the code generation process.

The build tool has been successfully implemented in various projects, including the development of flight controller software for hexacopters, quadcopters, battery controller software for a multilevel battery management system, and motor controller software development. Its adaptability and advantages are explored in depth in related research, facilitating standard-compliant software development. Additionally, the tool is applied in the development of flight control software for the PEGASUS project, focusing on an eVTOL UAV system, based on the incremental nonlinear dynamic inversion principle [SZM22]. This project will be published in 2024.

3 Advancements of the Build Tool

One important aspect of the software certification is the software architecture. DO-178C defines *software architecture* as: “The structure of the software selected to implement the software requirements"[DC11, Ri17]. Since Mrails did not have any kind of software architecture implementation previously and is based on MathWorks tools, it was justified to

use the System Composer. On the other hand, during the development of battery controller using Mrails for the hardware S32G274A, a need for integrating PIL capability was realized. This originated from the tremendous efforts that were needed to configure the Simulink models and test cases. To automate the execution of the requirements-based test cases in PIL mode, a new task is developed in Mrails.

3.1 Software Architecture using System Composer

In accordance with DO-178C compliance, traceability must be established between the software requirements and the software architecture. The architecture should encompass all the requirements and be feasible for implementation. To address the “big-freeze"problem and accommodate changes at a later stage, it is essential to create an architecture using a tool capable of handling updates seamlessly. Since the low-level requirements, in this case, the design models, are generated in Simulink, it enhances compatibility when linking architecture models created in System Composer and making frequent updates without the need to maintain the established traceability. As depicted in Figure 2, the process begins by creating software high-level requirements in Polarion, followed by the creation of the software architecture. This stage commences with the creation of an Mrails module in MATLAB via the command line, using an additional argument *‘-a’*. This argument assists the build tool in distinguishing between a software design module and an architecture module. The workflow from creating the software architecture module to its deployment is illustrated in Figure 4.

The architecture project path includes the customized Mrails profile, featuring stereotypes ‘module’ and ‘model’. Figure 3 shows a snapshot of a software architecture deployed for a flight controller for an eVTOL. The stereotypes are visible by the icons representing

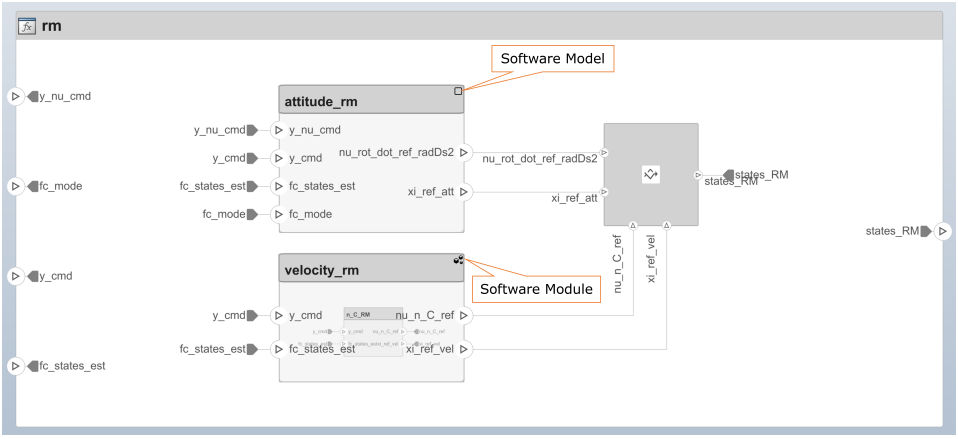


Fig. 3: Snapshot of a software architecture model with Mrails modules and models

modules and models. The subsequent step involves generating an architecture model (.slx) using the command *'create model-architecture'*. This command creates a placeholder software architecture model with the Mrails custom profile. Components are then added, and stereotypes are assigned based on specific requirements. A Mrails module is a software component which consists of several Mrails modules and/or models. For example, if a component is intricate enough to warrant a separate project, it is assigned the 'module' stereotype. This stereotype necessitates properties like sample time, module ID, and whether it is a top-level model. Components within a module can either serve as top-level models or belong to another module. In the latter case, the top-level module is labeled with the 'module' stereotype. Interfaces are linked to ports using the 'interface editor'. Software

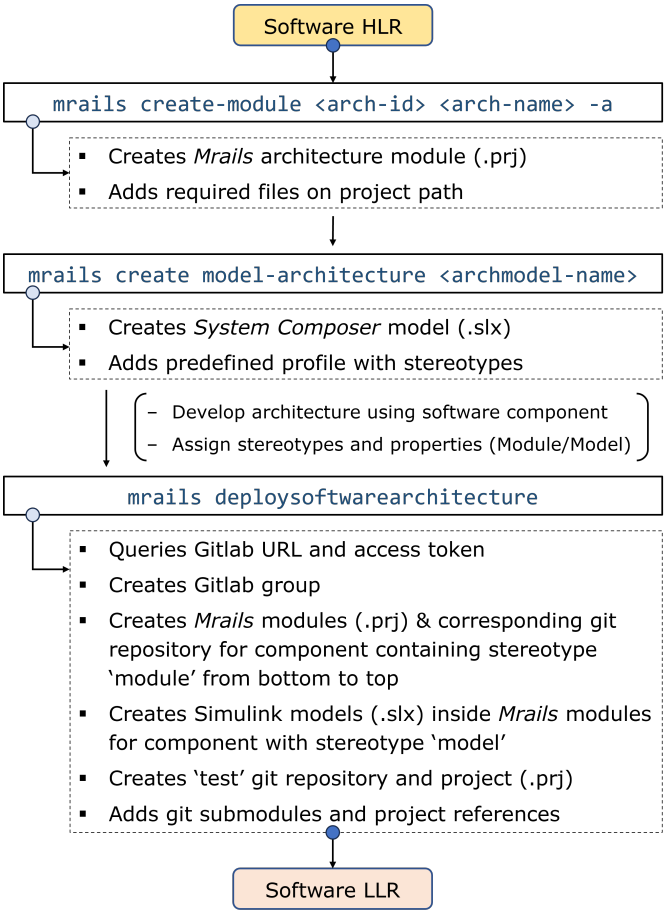


Fig. 4: Workflow of the software architecture development using the process-oriented build tool

components with the ‘model’ stereotype have properties specifying the model type, whether it’s reusable, a singleton, or a top-level model.

The next phase involves deploying the created software architecture. To achieve this, a task called *Deploy Software Architecture* is initiated. Upon execution, this task first retrieves Gitlab data such as URL, access token, and group name from the user. It then analyzes the architecture model, organizes it from the lowest to the highest level of the module hierarchy, and initiates a parallel pool in MATLAB. This pool creates the necessary Mrails modules and models, along with corresponding Git repositories and Gitlab projects with protected branches. Each Mrails module comprises two repositories: one for development and another for testing. The testing repository includes the development repository as submodules. Any

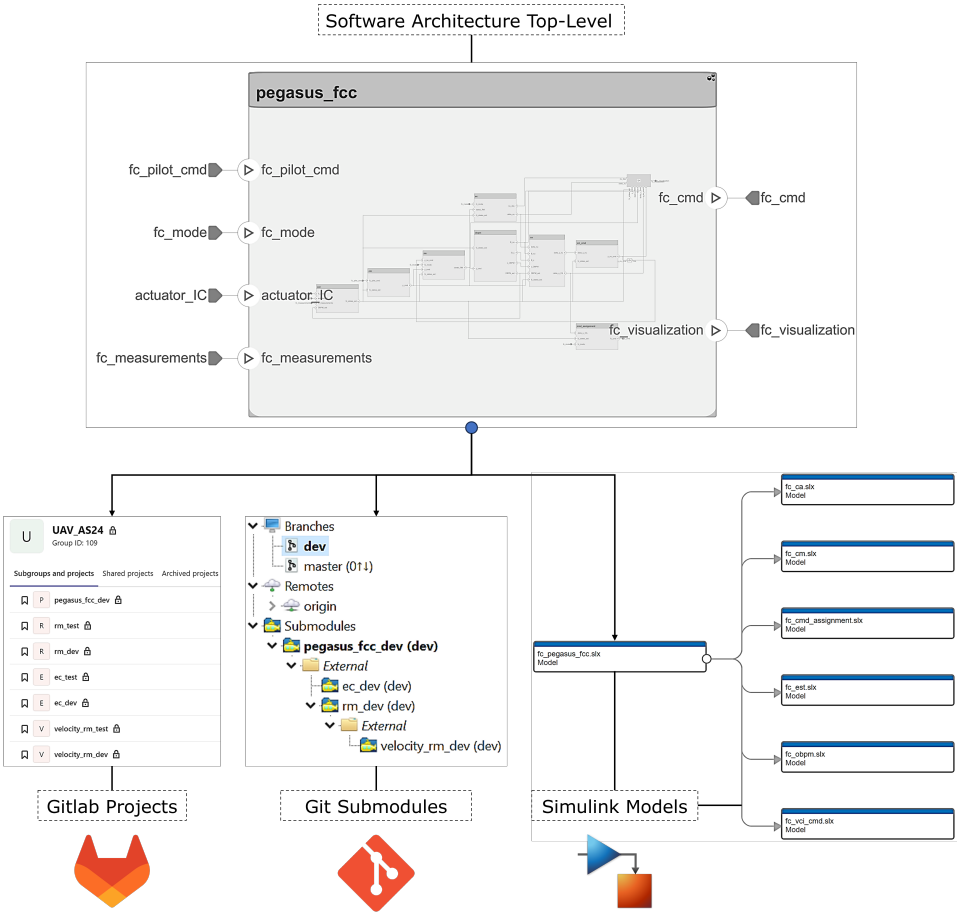


Fig. 5: Outcomes of the software architecture deployment process

Mrails module within another module is added as a Git submodule, as illustrated in Figure 4. Components with the 'model' stereotype are then generated using the command *create model-<top/singleton/reusable>*[Pa23b]. These models incorporate configuration settings in compliance with standards and project data dictionaries.

In this manner, the software architecture is not only deployed locally but also remotely with submodules. Figure 5 shows outcomes of the software architecture deployment process like Gitlab projects and respective Git submodules. The Gitlab projects contain MATLAB projects and derived Simulink models.

3.2 Processor-in-the-Loop Simulation using Trace32

Mrails already offers seamless automation for executing software-in-the-loop (SIL) simulations of requirements-based test cases. SIL is employed to assess the numerical equivalence between the model and the generated code. Executing requirements-based test cases in SIL simulation enables the calculation of an essential testing aspect—structural coverage.

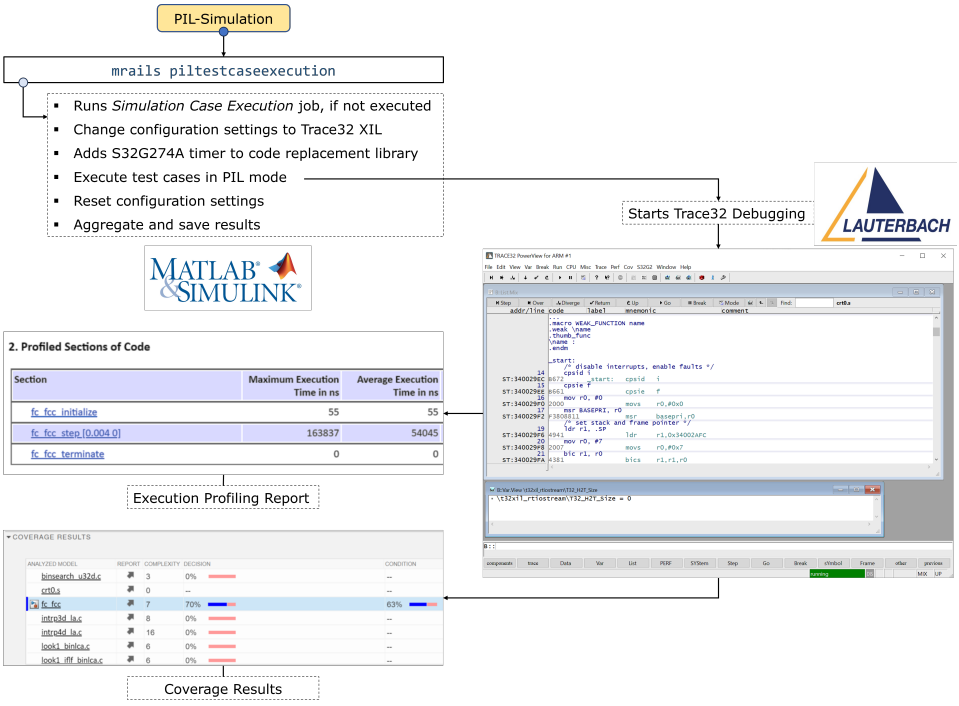


Fig. 6: Workflow of the Processor-in-the-Loop (PIL) Automation along with the build tool tasks and results

Mrails aggregates the results and presents them in an HTML web-based status report [Pa23c]. DO-178C specifies that structural coverage, when collected from the source code, is sufficient; however, it mandates a clear comparison between the source code and the object code.

During SIL testing, the source code is compiled for the host computer, which often differs from the production hardware. In PIL testing, the object code (instrumented) is cross-compiled for the production hardware and executed directly on the target in non-real-time. Simulink's PIL testing capability generates coverage results and code execution profiling. While some hardware, like the C2000 microcontroller [Ma23a], is readily supported, many targets require a custom target package. In such cases, Lauterbach Integration for Simulink [La23, Ma23b] comes into play. This tool facilitates running Simulink models directly on the target with the necessary configuration settings, producing coverage and execution profiling results.

Leveraging the framework provided by Mrails, the automation of PIL simulations for test cases has been achieved. The workflow for this task is illustrated in Figure 6. After verifying the test cases in normal simulation mode, PIL testing is conducted. The command *mrails piltestcaseexecution* is used to initiate the PIL testing task. To compare the numerical equivalence between normal simulation results and PIL results, it is crucial to check if the *Simulation Case Execution* job has been executed. Subsequently, the configuration settings of the models are adjusted to be compatible with the Trace32 XIL settings.

For execution profiling, a hardware timer is also necessary, and this feature is already integrated into the Mrails tool. The hardware timer for the M7 processor is added as a code replacement library in the configuration settings. The test cases are then executed in PIL mode, after which the Trace32 integration tool takes over. The results are sent back to the 'Test Manager' where they are compared with the simulation results to ensure equivalence.

4 Conclusion and Future Work

This paper outlines the new enhancements incorporated into a DO-178C process-oriented build tool. Among the two features introduced, one involves the implementation of software architecture, and the other pertains to the execution of PIL tests on S32G274A-M7 hardware. The research primarily delves into a specific component within the comprehensive DO-178C-based software development toolchain. To lay the groundwork for these enhancements, the software development toolchain is briefly introduced, followed by an overview of the Mrails build tool.

Enabling the implementation of software architecture and deploying this structure, both locally and remotely, offers several advantages to the team. Firstly, it ensures consistency since the structure is committed to Git. Secondly, the System Composer tool simplifies the linking of software requirements to the architecture model. As changes are inevitable, the

task analyzes whether the software component has already been deployed or requires action, proceeding accordingly.

It is essential to note that this feature is relatively new and holds potential for further improvement. Interfaces tend to undergo fewer changes than functionality, thus saving time and effort in creating separate software components for interfaces. The development, building, and verification of the interface module are considered future enhancements for this feature. To do so, a stereotype can be defined for the interfaces and ports. These interfaces can origin from interface management tool like dBricks. Another potential improvement involves adding additional stereotypes and properties to the profile to enhance flexibility. For instance, defining hardware stereotypes to automatically integrate the required configuration settings for specific hardware in the case of PIL simulation. Auto-commit and push practices run counter to the best practices of software development and can potentially disrupt remote repositories. To mitigate this, future plans include the addition of an extra argument to distinguish between auto-commit and manual commit.

Regarding the PIL automation feature described in this paper, the task essentially verifies the hardware node specified during project creation and, based on this, loads the relevant configuration settings onto the project. Currently, only S32G274A-M7 hardware is supported, but future expansion to accommodate other hardware is envisaged. During this research, an issue arose when enabling both coverage settings and execution profiling simultaneously. The cause of this issue remains unknown, and it constitutes an area for future investigation.

Managing the design and code configuration process is very tedious and leads towards inconsistency during the software development process. This mitigates to the verification phase and then finally creates a ruckus at integration level. The advancements presented in this research are a stepping stone towards automating the software development process and reducing human errors resulting into fast and efficient software development. Although the advancements do not directly achieve the DO-178C/DO-331 objectives but presents an efficient way to achieve them. For example, the software architecture can also be developed as simple diagram in a tool like Polarion, however, there is no such feature to deploy the architecture directly to Simulink. Deployment of the architecture models to Simulink include creation of the project, interfaces in data dictionary and template models.

In the context of PIL testing, no such automation task was available till now which can configure the model settings according to the target and run the requirements-based test cases on the target. The build tool Mrails is also continuously being improved in other areas, with new features constantly being added, such as recursive execution of jobs, accumulating code metrics, and implementing continuous integration server to test the development of the tool and its applications. Parallel execution of jobs and automation of possible manual checks are some of the future development topics. The feature presented in this paper to automate the PIL testing has now made it possible to run extensive tests on continuous integration server. This research has aimed at facilitating the use of model-based software development technology and leverage its advantages when seeking for certification along

the DO-178C/DO-331 standards. The process-oriented build tool is currently being applied to develop battery controller, motor controller and flight controller for electric motor glider and eVTOLs.

Acknowledgements

This research and project ELAPSED are funded by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr. dtec.bw is funded by the European Union – NextGenerationEU [UotB21].

Bibliography

[Cl21] Cleland-Huang, Jane; Agrawal, Ankit; Vierhauser, Michael; Mayr-Dorn, Christoph: Breaking the Deep Freeze: Visualizing Change in Agile Safety-Critical Systems. IEEE Software, 38(3):43–51, May 2021.

[dB23] dBricks: , dBricks — peerss.ru. https://peerss.ru/en/dbricks/, 2023. [Accessed 27-10-2023].

[DC11] DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Standard, RTCA, 2011.

[Dm20] Dmitriev, Konstantin; Zafar, Shanza Ali; Schmiechen, Kevin; Lai, Yi; Saleab, Micheal; Nagarajan, Pranav; Dollinger, Daniel; Hochstrasser, Markus; Holzapfel, Florian; Myschik, Stephan: A Lean and Highly-automated Model-Based Software Development Process Based on DO-178C/DO-331. In: 2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC). pp. 1–10, 2020.

[DO11] DO-331: Model-Based Development and Verification Supplement to DO-178C and DO-278A. Standard, RTCA, 2011.

[HMH19] Hochstrasser, Markus; Myschik, Stephan; Holzapfel, Florian: Application of a Process-Oriented Build Tool for Flight Controller Development Along a DO-178C/DO-331 Process. In: Communications in Computer and Information Science. Springer International Publishing, 2019.

[La23] Lauterbach: , Microprocessor Development Tools. https://www.lauterbach.com/, 2023.

[Ma23a] MathWorks: , PIL Simulation - MATLAB & Simulink — mathworks.com. https://www.mathworks.com/help/ti-c2000/ug/pil-simulation.html, 2023. [Accessed 26-10-2023].

[Ma23b] MathWorks: , TRACE32 PowerView — mathworks.com. https://www.mathworks.com/products/connections/product_detail/trace32-powerview.html, 2023. [Accessed 26-10-2023].

[NX23] NXP: , S32 Design Studio IDE — nxp.com. https://www.nxp.com/design/software/development-software/s32-design-studio-ide:S32-DESIGN-STUDIO-IDE, 2023. [Accessed 27-10-2023].

[Pa22a] Panchal, Purav; Myschik, Stephan; Dmitriev, Konstantin; Bhardwaj, Pranav; Holzapfel, Florian: Handling Complex System Architectures with a DO-178C/DO-331 Process-Oriented Build Too. In: 2022 IEEE/AIAA 41st Digital Avionics Systems Conference. 2022.

[Pa22b] Panchal, Purav; Myschik, Stephan; Dmitriev, Konstantin; Holzapfel, Florian: Application of a Process-Oriented Build Tool to an INDI-Based Flight Control Algorithm. In: AIAA AVIATION 2022 Forum. American Institute of Aeronautics and Astronautics, June 2022.

[Pa22c] Panchal, Purav; Sorokina, Nina; Myschik, Stephan; Dmitriev, Konstantin; Holzapfel, Florian: APPLICATION OF A PROCESS-ORIENTED BUILD TOOL TO THE DEVELOPMENT OF A BM3 SLAVE CONTROLLER SOFTWARE MODULE. In: DGLR DLRK 2022 Forum. DGLR, September 2022.

[Pa23a] Panchal, Purav; Bliemetsrieder, Wolfgang; Sorokina, Nina; Myschik, Stephan: Real-Time Verification of A Battery Slave Controller Developed Using a DO-178C/DO-331 Based Process-Oriented Build Tool. In: AIAA AVIATION 2023 Forum. American Institute of Aeronautics and Astronautics, June 2023.

[Pa23b] Panchal, Purav; Dmitriev, Konstantin; Myschik, Stephan; Holzapfel, Florian: Comprehensive Overview of a Process-Oriented Build Tool for Airborne Safety-Critical Software Development. In: 2023 10th International Conference on Recent Advances in Air and Space Technologies (RAST). IEEE, June 2023.

[Pa23c] Panchal., Purav; Sorokina., Nina; Kuder., Manuel; Myschik., Stephan; Dmitriev., Konstantin; Holzapfel., Florian: Application of a Process-Oriented Build Tool for Verification and Validation of a Battery Slave Controller for a Battery Modular Multilevel Management System Along the DO-178C/DO-331 Process. In: Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering - MODELSWARD,. INSTICC, SciTePress, pp. 184–193, 2023.

[Ri17] Rierson, L.: Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance. CRC Press, 2017.

[Si04] Siemens: , Application Lifecycle Management (ALM), Requirements Management, QA Management | Polarion - Software — polarion.plm.automation.siemens.com. https://polarion.plm.automation.siemens.com/, 2004.

[SZM22] Surmann, Denis; Zrenner, Maria; Myschik, Stephan: Flight Performance Evaluation of a Conceptual eVTOL System using Nonlinear Simulations. In: AIAA SCITECH 2022 Forum. American Institute of Aeronautics and Astronautics, January 2022.

[UotB21] University of the Bundeswehr, Munich: , ELAPSED — unibw.de. https://www.unibw.de/elapsed, 2021. [Accessed 26-10-2023].

[Ve23] Vector: , VectorCAST - Software Test Automation for High Quality Software. https://www.vector.com/int/en/products/products-a-z/software/vectorcast/, 2023. [Accessed 27-10-2023].

Case Study: Securing MMU-less Linux Using CHERI

Hesham Almatary[1] Alfredo Mazzinghi[2] Robert N. M. Watson[3]

Abstract:

MMU-less Linux variant lacks security because it does not have protection or isolation mechanisms. It also does not use MPUs as they do not fit with its software model because of the design drawbacks of MPUs (i. e. coarse-grained protection with fixed number of protected regions). We secure the existing MMU-less Linux version of the RISC-V port using CHERI. CHERI is a hardware-software capability-based system that extends the ISA, toolchain, programming languages, operating systems, and applications in order to provide complete pointer and memory safety. We believe that CHERI could provide significant security guarantees for high-end dynamic MMU-less embedded systems at lower costs, compared to MMUs and MPUs, by: 1) building the entire software stack in pure-capability CHERI C mode which provides complete spatial memory safety at the kernel and user-level, 2) isolating user programs as separate ELFs, each with its own CHERI-based capability table; this provides spatial memory safety similar to what the MMU offers (i. e. user programs cannot access each other's memory), 3) isolating user programs from the kernel as the kernel has its own capability table from the users and vice versa, and 4) compartmentalising kernel modules using CompartOS' linkage-based compartmentalisation. This offers a new security front that is not possible using the current MMU-based Linux, where vulnerable/malicious kernel modules (e. g. device drivers) executing in the kernel space would not compromise or take down the entire system. These are the four main contributions of this paper, presenting novel CHERI-based mechanisms to secure MMU-less embedded Linux.

Keywords: Linux; CHERI; security; memory safety; compartmentalization; embedded systems; operating systems

1 Introduction

Linux is the most widely deployed Operating System (OS) in the world. It is the base of Android, cloud computing, and millions of Internet of Things (IoT) devices. The current MMU-less embedded Linux variant lacks security because either the underlying processor does not have an Memory Managment Unit (MMU) or it does have it but some systems intentionally do not make use of it due to determinism, power, and simplicity requirements

[1] Capabilities Limited, U.K. heshamalmatary@capabilitieslimited.co.uk
[2] Capabilities Limited, U.K. alfredo@capabilitieslimited.co.uk
[3] Capabilities Limited, U.K. robert@capabilitieslimited.co.uk

doi:10.18420/sw2024-ws_06

(typically in real-time systems). There exists other protection technologies such as TrustZone and Memory Protection Unit (MPU)s, but they are still not widely-used in MMU-less Linux because of their coarse-grained security features [Zh19] that are not suitable for embedded Linux. Capability Hardware Enhanced RISC Instructions (CHERI) tries to fill this gap by providing fine-grained security at lower overhead. In this paper, we try to enhance MMU-less Linux's security by having three main goals: first, complete spatial memory safety at the pointers level, and second, spatial compartmentalisation of user programs, libraries, and kernel modules. Pointer safety protects against 70% of software vulnerabilities [Ci19] (even in MMU-based systems), while compartmentalisation helps limit the "future, unknown" attack effects to the compromised compartment instead of the entire system. The third goal is source-code compatibility which ensures minimum to no changes to the C code. This is vital for high-end, MMU-less embedded systems that are already written in millions of Lines of Code (LoC) like Linux and its userspace. We achieve these goals by building on CHERI as the sole hardware feature to secure MMU-less Linux with four different approaches:

1. Building the entire software stack in CHERI C to provide complete spatial pointer safety at the kernel and user space.
2. Isolating user programs from each other by restricting access for each ELF to what it can reference through a restricted capability table.
3. Isolating user programs from the kernel by taking away system permissions from user capabilities and having separate capability tables for the kernel and user applications.
4. Compartmentalising kernel modules within the kernel itself (which cannot be realised even with an MMU), by implementing the CompartOS model [Al22a, Al22b].

We evaluate these approaches by porting MMU-less Linux to CHERI-RISC-V [Al23b] (a CHERI implementation on top of the RISC-V hardware and software ecosystem) along with userspace that contains *Busybox* [Al23a], a simple run-time linker called *uldso* [Al23d], and the *uclibc-ng* [Al23c] C library. All of our work is open-source. We report our experiences porting MMU-less Linux and its components to CHERI and applying the CompartOS compartmentalisation model [Al22a, Al22b] to isolate user programs and kernel modules.

The porting is still a work-in-progress and is not covering the entire Linux kernel, configurations, and code execution paths. However, it is a best-effort starting point with minimal configurations enabled that are able to run *Busybox*'s shell utilities, load simple kernel modules, and run user applications, all in CHERI C (pure-capability mode, which guarantees complete spatial memory safety). This could serve as a future reference or base for those who want to secure other (embedded or MMU-less) operating systems using CHERI. We hope this paper gives insights to what it takes to secure complex and high-end embedded operating systems using CHERI, and the most common software subsystems that require modifications.

2 Background

In this section, we give a brief overview of CHERI and what security challenges it is trying to address, mainly pointer and memory safety and software compartmentalisation. We also describe capability-based security that CHERI brings and its advantages over traditional Access Control List (ACL) and MMU-based systems that exist in UNIX systems (including Linux).

2.1 Memory Safety

Memory-safety software vulnerabilities have arguably been the most prevalent type of bugs in the history of computers and software in general. In their Eternal War in Memory paper [Sz13], Szekeres et al. argue that memory corruption bugs are one of the oldest problems in computing and still are, regardless of the efforts spent to come up with memory-safe languages and hardware security architectures. Stack and buffer overflows [On96] have been one of the most crucial bugs that can be exploited to affect the integrity, confidentiality, and availability of a computer system. Memory safety is thus a vital attack vector in both general-purpose OSes and embedded software systems that are mostly written in memory-unsafe languages such as C/C++ for performance and fine-grained hardware control purposes. For example, Microsoft has recently revealed [Ci19] that 70% of security software bugs in their systems (e. g. Microsoft Windows written in C/C++) are memory-safety related, which include spatial and temporal memory safety. This issue does not appear to be only specific to general-purpose OSes such as Windows, but also in the embedded software systems. Papp et al. have done a sound security analysis in embedded systems [PMB15] and found out that memory-safety and programming error attacks on embedded operating systems and firmware are the most frequent and critical.

Some countermeasures against memory safety attacks in embedded systems have been proposed [PW08]. These include hardware architecture support, static and dynamic analysis tools, compiler support, the use of memory-safe languages, and software compartmentalisation. For example, guard pages [Co98] rely on the MMU to detect buffer overflows in general-purpose OSes. As our target OSes do not usually use the MMU, they tend to go for lighter-weight software stack protection solutions. For instance, stack canaries [Wi22a] can be optionally enabled by the compiler toolchain to detect stack overflows. Similarly, some embedded operating systems such as RTEMS and FreeRTOS implement their own stack overflow detection in software [Frb, RTa]. However, such solutions are coarse-grained (i. e. only protect the stack of a task, rather than smaller buffers or global objects) and can be bypassed [Ri02].

2.2 Software Compartmentalisation

Software compartmentalisation [Wa15, Gu15, Ka87, PFH03, Ki03, Wa10] is a technique to split up a large monolithic software into smaller compartments in order to reduce the attack vector and limit the effects of a successful attack only to the compromised compartment. Unlike vulnerability mitigation techniques, software compartmentalisation assumes that zero-day unknown vulnerabilities always exist and acts accordingly. The measures that are taken to apply software compartmentalisation follow the *principle of least privilege* [SS75] by only giving the very minimum privileges to each compartment required to perform a service. This reduces privilege escalation attacks due to the *ambient authority* problem [La73] and, thus, maintains the *integrity* and *confidentiality* of the whole system.

Traditionally, OSes compartmentalise applications into threads or processes. Each could be isolated from each other by the MMU or MPU. However, the recent attacks require more scalable and fine-grained form of compartmentalisation within the same address space (e. g. in a single process or kernel). This could be linkage modules (e. g. kernel modules, device drivers, or third-party libraries) or even down to each pointer or function. Compartmentalising such smaller components within the same address space not only includes spatial and temporal memory safety, but also fault isolation (to maintain availability).

2.3 Capability-based Systems

A capability, in general, is an unforgeable token to an object in the system that authorises its holder to access that object with a set of permissions embedded in the capability itself. Thus, a capability serves as both an identification (unlike ACL systems e. g. UNIX in which identities are separate from resources in the access lists) and an authorisation mechanism within a protected system. The capability system itself is only responsible for the integrity of capabilities (i. e. they cannot be forged) and serving requests to create, copy, and revoke capabilities. The properties of a capability give it some advantages over the ACL systems. Capability-based systems address some of the issues like the size of the table and the requirement to have a list of resources for each domain, which ACL systems fail to achieve. Furthermore, capabilities inherently adopt the notion of *intentionality*; an entity (such as a process) that has a capability to an object is only allowed to access this object with limited access permissions to do its job. This solves challenging issues in ACL systems such as the *confinement* [La73] and *confused deputy* [Ha88] problems. Overall, capability-based systems give some practical solutions to security and protection issues such as, *the confused deputy, the confinement problem, scalable access control management and fine-grained access control*.

There have been significant efforts to build capability-based systems in software, hardware, programming languages, or a combination of them; most of which are introduced in Levy's capability-based systems book [Le84]. Hydra [Wu74] was the first general-purpose object-based capability system. It was developed at Carnegie Mellon University. The primary

motivation for Hydra was to allow operating systems research and extensibility. Some of the design choices like putting drivers in userspace and separating policies from mechanisms in the kernel found their way to microkernel designs [Li95] and are still being adopted in modern L4 microkernels such as seL4 [Kl09] and Fiasco.OC [L4]. Hydra provided a new system abstraction at the time, making everything in a computer system an object. EROS [SSF99] introduced a new idea of revoking capabilities by versioning objects and capabilities pointing to them. To revoke access to an object, the version of the object would be changed, and consequently, the mismatched versions (during a dereference) will trigger a fault. Capabilities in EROS are represented as nodes. A protection domain consists of a tree of nodes of capabilities. Each node has a fixed size of thirty-two capabilities. Unlike seL4 (described next), EROS tries to map pages on virtual memory faults, and if it fails, it calls a user-level fault handler in a capability. seL4 always redirects faults to the fault handler in userspace. seL4 is a modern microkernel with a focus on security within embedded systems. The authors of the 2009 seL4 paper claimed it is the first general-purpose operating system to be formally verified [Kl09]. This means that the high-level kernel specification matches the C code (and in later versions, the binary itself). With the assumption that the assembly code and the hardware are correct, seL4 is claimed to be bug-free and would never crash. This comes with a few caveats and assumptions [sP]. For instance, seL4 does not currently protect against timing channels, DMA-related attacks, or further exploitable hardware vulnerabilities. It also only proves the integrity and confidentially of the system with those caveats. Adopting microkernel principles, seL4 embraces simplicity in its design and implementation. This enabled the trusted codebase to be small enough (less than 10K LoC) that complete formal verification of the kernel behaviour in every possible path is performed and reasoned about to be bug-free. This effort narrowed the gap of having a trustworthy software system. The downside, still, is that the hardware is assumed to be correct, which is not usually the case. For example, the verified seL4 code is vulnerable to recent covert channel attacks such as Spectre [Ko18]. Furthermore, seL4 relies on conventional MMU to provide isolation, which largely lacks determinism and have coarse-grained memory protection granularity of 4 KiB. That is, seL4 will not be a good fit for embedded systems that require determinism and small size protection units (e. g. 4 bytes Memory Mapped Input Output (MMIO) registers).

2.4 CHERI Overview

CHERI is a modern capability-based Reduced Instruction Set Computer (RISC) architecture that is being developed by the University of Cambridge and SRI International. It is both hardware (as an Instruction Set Architecture (ISA) extension) and software (toolchain, programming languages, operating systems, and applications).

The main principles CHERI is designed around are:

- *Principle of least privilege*: which motivates the idea of reducing privilege rights to a piece of software as much as possible.

- *Principle of intentional use*: by naming the privilege a software application uses, rather than giving it full privilege accesses and let it choose what privileges to use implicitly.

CHERI is designed with these principles in mind such that a software application that runs on top of CHERI inherently maintains capability attributes. This helps in reducing the access rights an attacker has and consequently minimises the attack surface. There are two main protection models in CHERI: pointer safety and software compartmentalisation. Pointer safety is the primary application of CHERI in C/C++ languages. This mostly relies on the compiler toolchain to map C/C++ pointers into CHERI's memory capabilities. In the current implementation of CHERI, the toolchain is LLVM/Clang. There are two main modes a CHERI-aware C/C++ code can be compiled in: compatible (or hybrid) and pure-capability modes (or simply CHERI C). Hybrid mode enables users to manually select pointers to protect, while CHERI C automatically protects all pointers.

The CHERI hardware uses a fat-pointer representation for capabilities, along with a hardware-managed tag bit that determines whether the capability is valid. In particular, a CHERI capability encodes the *base* and *top* of the region in which the capability can be dereferenced. A number of permission bits determine how the capability can be used (e. g. read-only, read-write, etc.). Tagged memory is used to maintain the tag bit for capabilities stored in memory. The architecture uses the tag bit to enforce the CHERI capability integrity and provenance validity properties. The ISA maintains the capability monotonicity property, whereby instructions can only narrow the permissions and bounds of any given capability. CHERI hardware capabilities form the primitive upon which it is possible to implement language-level spatial memory safety and software compartmentalisation models.

Apart from pointer safety, CHERI provides the flexibility for software developers to define their own representation of a software compartment in order to logically split large monolithic software systems into smaller compartments. For example, CHERI could be used to compartmentalise processes, linkage-modules, static or shared libraries, OSes and applications, etc.

	MMU	**CHERI/CompartOS**
Pointer safety	✗	✔
Compartmentalisation	✔	✔
Virtualisation	✔	✗
Capability-based protection	✗	✔
In-address-space isolation	✗	✔
Protection granularity	4 KiB	1 Byte
Isolated memory resources	User processes and kernel	User processes, kernel, libraries, kernel modules, and pointers.

Tab. 1: MMU vs CHERI comparison.

CHERI is fundamentally different than MMU. MMUs are implemented in hardware and managed by the OS to provide both protection and virtualisation at 4 KiB granularity. CHERI, on the other hand, is an ISA extension that is implemented also in hardware, but could be managed by the OS, programming languages, linkers and loader, and applications in order to provide fine-grained pointer safety and memory protection (e. g. 1 byte) and scalable software compartmentalisation (e. g. isolating processes, libraries, or modules). The main differences between CHERI and MMUs are shown in Table 1. In this paper, we deploy two main applications of CHERI: 1) fine-grained memory safety in C/C++, and 2) software compartmentalisation.

CHERI is capable of replacing the MMU in terms of protection and isolation between processes. However, this is only one contribution of this paper (contribution number 2 in Section 1). CHERI and CompartOS offer more security features than MMUs. MMUs cannot provide pointer and memory safety at the programming languages level as CHERI does because of their coarse-grained page size granularity. Furthermore, CHERI and CompartOS allow in-address-space protection; two different pointers, user applications, or kernel modules can be isolated from each other. Arguably speaking, this cannot be done via MMUs unless the entire design and implementation of the Linux kernel and applications are going to be changed. As shown later, it takes minimal effort to do so using CHERI/CompartOS in terms of LoC changes, embracing source-code compatibility. All in all, CHERI/CompartOS provide more security, scalability, compatibility in embedded systems against recent attacks even compared to MMUs.

2.5 MMU-less Linux

MMU-less Linux is a variant of the mainstream Linux targeting embedded applications. It is a configuration option to enable building and running smaller and customised Linux subsystems. Systems choose not to use the MMU for two main reasons: 1) either the underlying hardware does not have an MMU unit, or 2) there exists an MMU, but managing it adds extra complexity, size, or does not meet some requirements such as real-time, determinism, power consumption, etc. MMU-less Linux could have coarse-grained privilege-separation between the kernel and user. However, there is no spatial memory protection among user programs, kernel modules, and the kernel. MMU-less Linux uses light-weight binary formats for ELFs. Historically, FLAT-ELF was mostly used but comes with restrictions such as limited number of shared libraries, and no support for dynamic loading via *dlopen*. ELF-FDPIC format tries to overcome such limitations and gets all of the usual ELF features, but the code has to be all position independent (PIC). This allows different load segments to be independently located in memory while still being able to share the text segment, but not necessarily data segments. In this paper, we use RISC-V as a base for MMU-less Linux, which does not use MMU or MPU/PMP. We further choose ELF-FDPIC which works best for our CHERI-based compartmentalisation in embedded systems.

2.6 Potential Applications in Avionics

While CHERI and CompartOS can be applied to mainstream baremetal and tiny embedded systems (at a cost), it aims to secure the high-end range of *mainstream* embedded operating systems (including Real Time Operating System (RTOS)es), such as Linux. By mainstream, we mean traditional embedded (operating) systems that are mostly unprotected and shy away from using MPUs and MMUs as they do not meet their requirements. This is in contrast to using *new* security architectures that target small systems (e. g. TockOS [Le17] and ACES [Cl18]) and require existing applications to be (re)written on top of them. Traditional embedded operating systems are facing security concerns and need some form of protection while continuing to meet their real-time and safety-critical requirements such as partitioning, bounded processing, high determinism and high throughput. Developers either try to use the MPU in MCUs that provide it, but that is not scalable or fine-grained enough for complex applications, or they go for general-purpose processors (e. g. A-class Arm processors) that have MMUs (that are not frequently used due to complexity and performance non-determinism) and rather use the MPU. The area those systems fit in is safety-critical avionics and automotive. For example, FreeRTOS or RTEMS run on Raspberry Pi [Wi22b] or Beagle [Wi21a] boards (embedded with Arm A-class processors) without an MMU-based process or protection model. CompartOS mainly targets the low-end A-class category. Example deployed systems are:

- Amazon's FreeRTOS with WiFi, TCP/IP, and Bluetooth stacks running on high-end M-class processors (Amazon Web Services (AWS) Reference Integrations [Fra]) and used for feature-rich IoT.
- Primus Epic Avionics: Deos and ARINC 653, running on x86 and Arm's A-class [DI].
- VxWorks CERT EDITION, running on NXP QorIQ [Wi21c] for automotive and avionics [WRS].
- RTEMS used in NASA's Magnetosphere Multiscale (MMS) Mission [NA, RTb] running on the Coldfire CPU [Wi21b].

On the application use cases, some safety-critical standards such as ARINC 653 could greatly fit with the CompartOS model. Most ARINC 653 implementations, however, are proprietary and closed source. While we mainly apply CHERI and CompartOS models to Linux in this paper, we believe that they have been also applied to other avionics systems such as RTEMS and FreeRTOS, and could easily be applied to VxWorks and ARINC 653 implementations to provide partitioning, bound processing, and availability.

3 Design

The main motivation for this work is to take an unsecure monolithic MMU-less Linux and secure it using CHERI as in Figure 1. In an MMU-less Linux, there is only privilege-level

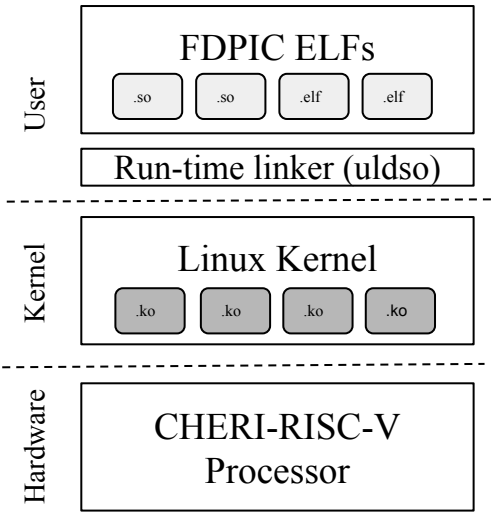


Fig. 1: Secured MMU-less Linux using CHERI. All build and run in CHERI C. Grey boxes represent different types of compartments such as kernel modules (.ko) and ELF-FDPIC programs (.elf and .so).

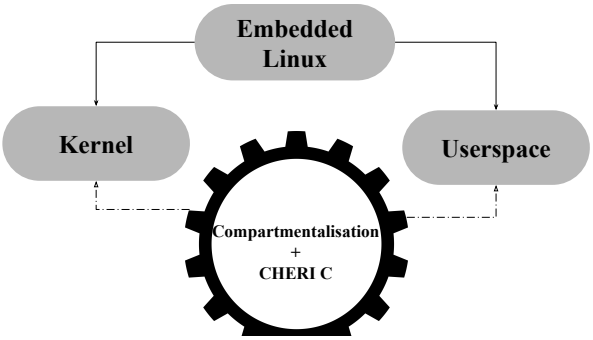


Fig. 2: The process of securing MMU-less Linux using CHERI is divided up into CHERI C and compartmentalisation stages for each software component.

separation between the kernel and userspace. There is no memory safety or separation within or among user applications or kernel modules. Malicious user applications or modules can spy on (no confidentiality) or affect the functionality of one another (no integrity), or take down the entire system (no availability). CHERI-based memory safety and compartmentalisation provide some form of confidentiality, integrity, and availability.

The process of doing that is divided up into steps shown in Figure 2. Securing MMU-less Linux is done in two steps: *CHERIfying* the code and enforcing a compartmentalisation model. *CHERIfying* means that we port the existing C code into CHERI C (or pure-capability mode) where every pointer is a capability. The source-code has to be available to re-compile. This provides complete spatial memory safety within the entire software stack. It helps with the confidentiality and integrity of the software system. However, it does not necessary help with availability; a CHERI security violation in the kernel may still bring down the whole system. That is where compartmentalisation is important. Compartmentalisation divides up the entire monolithic kernel (and userspace) into smaller compartments; each compartment is contained. A fault in one compartment does not affect the rest of the system. This is crucial for third-party (and likely untrusted) user libraries and kernel modules such as device drivers.

For CHERI C, each software subsystems in the kernel and userspace needs modifications in the following areas:

1. **Build system:** necessary changes to provide the proper toolchain flags to enforce CHERI C memory safety.
2. **Low-level**: mostly architecture-dependent code such as booting, handling traps, atomics, etc.
3. **Pointers**: some/most C/C++ projects mix using integer types (e. g. *int* and *long*) to store both integers and pointers and do computations and arithmetic on them, as this is allowed by C. This could lead to out-of-bounds memory safety bugs. CHERI C is strict and it differentiates between normal integers and address pointers as CHERI capabilities.
4. **Provenance**: every capability has to be derived from another valid capability with the same or less bounds and permissions (monotonicity attribute). There are subsystems in the OS that need to manually create capabilities. For instance, the boot code creates capabilities for global objects and functions from ***DDC*** (root data capability) and ***PCC*** (root code capability), respectively. Similarly, new capabilities for MMIO need to be created for device drivers.
5. **Allocators**: dynamic memory allocators (e. g. *malloc()*) need to be modified to return valid bounded capabilities with the correct permissions. Extra changes could also be applied to ensure temporal memory safety (e. g. revocation, memory zeroing, etc).

For compartmentalisation, the following subsystems need to be changed:

	Linux Kernel	uldso	uclibc-ng	Busybox
Build system	3/3	1	0	6
Low-level	1926/893	16	309/67	5
Pointers	324/280	0	0	0
Syscalls	553/543	0	37/37	0
Provenance	26/13	0	0	0
Misc	46/36	0	219	0
Compartmentalisation	468/28	80	0	0
Allocators	0	0	18	0

Tab. 2: Number of inserted/deleted lines to secure embedded MMU-less Linux using CHERI.

1. **Build system**: necessary changes to provide the proper toolchain flags to enforce CompartOS compartmentalisation for compartments.
2. **Start-up**: to be able to build a capability table for each compartment representing its protection domain and interface with other compartments.
3. **Cross-compartment calls**: to create capabilities that perform inter-compartment calls, grant them to callers, and emit necessary trampolines to perform compartment and protection domain switches.
4. **Fault handling and recovery**: an implemented custom policy to handle CHERI C security violations per compartment.

In the following section, we discuss the implementation details of the specific MMU-less Linux software subsystems we worked with.

4 Implementation

We have used the upstream Linux (version 6.1) and the RISC-V port without MMU as a baseline. Userspace consists of *Busybox* [Bu] and a simple run-time linker [Unb], along with *uclibc-ng* [una] as a user C library. We run the complete software stack on *QEMU*. This gets us a shell and all *Busybox* utilities. Table 2 demonstrates the modified systems and the LoC changes for each.

4.1 Kernel

Low-level: The boot code which is usually written in assembly is the first subsystem that needs to be modified. First, ***PCC*** and ***DDC*** permissions need to be restricted (i. e. not to include execute and/or load/store permissions). Furthermore, the instructions and registers need to be modified to deal with capability registers rather than integer registers. This

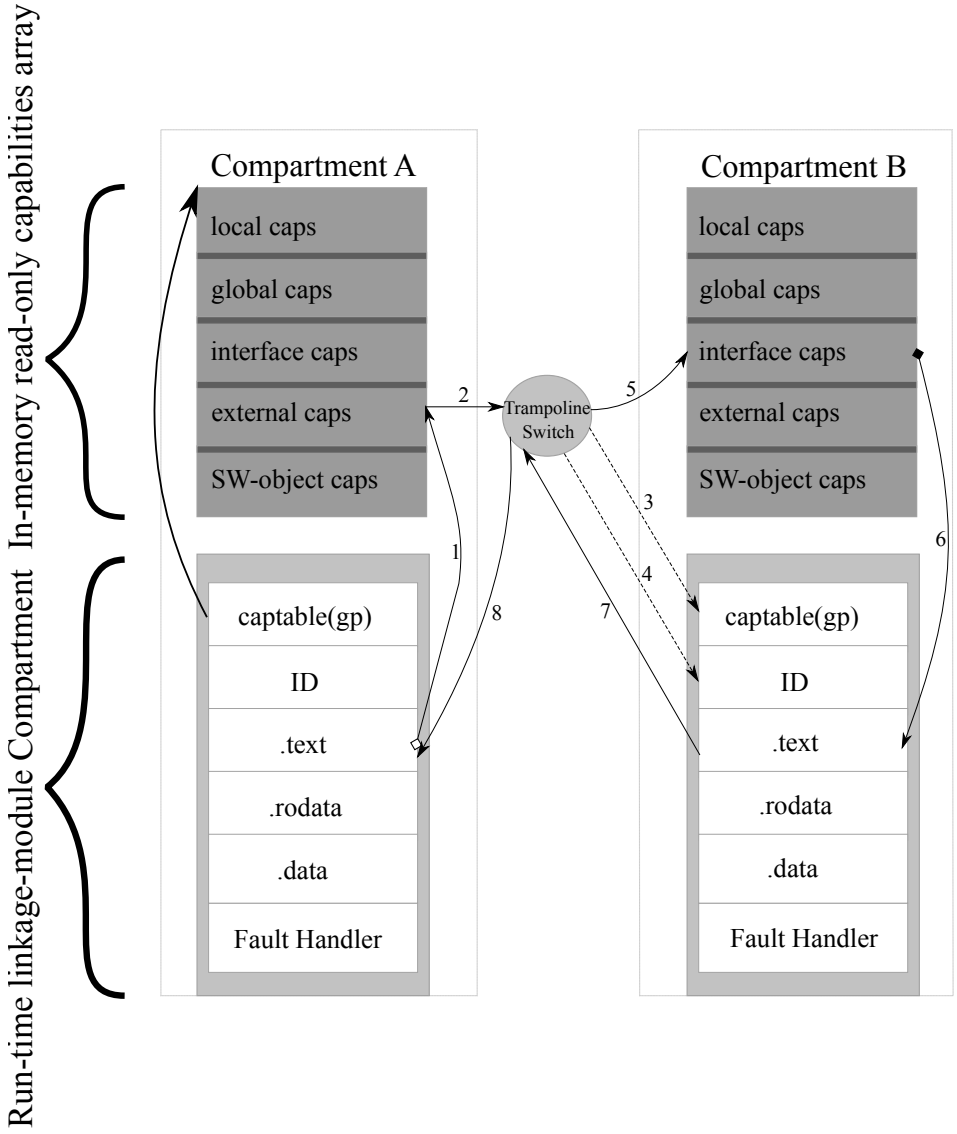


Fig. 3: A runtime example of a compartmentalised Linux kernel system and a protection domain switch. Each compartment is this case is a kernel module. Filled lines are capability references, while dashed lines are operations. Numbered edges denote the sequence to perform an external call to another compartment, which triggers a domain switch. The switch starts with a function call that references a capability from the externals capability list (#1). If the external capability is present and valid, it points to a small, read-only trampoline (#2) that performs compartment switches by setting the new captable and compartment ID (#3 and #4) after storing the caller's context. It then jumps via the interface capability (#5) provided by the target compartment. This interface capability points to the function within its associated compartment (#6). Upon its return (#7), the trampoline restores the caller's context, captable and ID, then returns back to the caller function (#8).

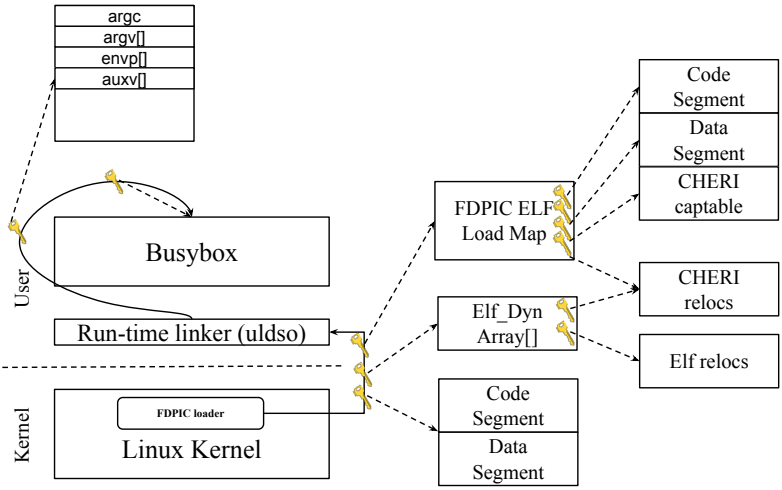


Fig. 4: A runtime example of a compartmentalised Busybox userspace. The keys are capabilities. Solid lines/curves represent program transfer flow. Capabilities on these lines are passed as register arguments. Dashed lines show what object each capability holds an authority to.

includes atomic instructions as well. The code starts in hybrid mode, then the capability table is written for all the global pointers in the Linux kernel. These are driven from ***PCC*** (for code capabilities like functions) and ***DDC*** (for data capabilities like variables, arrays, and any other data objects). The exception table and trap register also need to be set up to be capability-aware. Finally, the kernel jumps to the C code in pure-capability mode. Exception code is also written in assembly, but only instructions and registers need to be modified to deal with capabilities. There are other higher-level changes in the architecture dependent code, still. For example, the types and sizes of the registers, pointer variables, stack pointer, program counter, return address, frame pointers, etc., all needed to be capabilities instead of integers. Similar changes in the memory management code that converts Page Frame Number (PFN) to virtual/physical addresses also needed to be return capabilities.

System calls: Generic Linux system calls macros treated all arguments and return values as type *long*. This needed to be changed to be pointers or capabilities. There are some specific system calls (used by *Busybox*) that also needed to be changed. For instance, *ioctl* system calls and their kernel functions receive an argument from the user that could hold pointers. Even though we changed the lowest level of the system call macros to receive capabilities, further *ioctl* handler functions still had the argument type as *long*. This was a bit disruptive

```
xPortCompartmentEnterTrampoline:
% Metadata
.Lfunc:       .zero CAP_SIZE % Callee's function capability
.Lcaptable:   .zero CAP_SIZE % Callee's capability table
.Lcompid:     .zero CAP_SIZE % Callee's compartment ID
// Compartment/Domain Switch
Save caller's context (register set, GP, compid)
Setup new GP/captable
Set currentCompartmentID = Lcompid
(Optional) Restrict/bound the callee's stack
Call the destination function
Restore caller's context (register set, GP, compid)
Return to the caller compartment
```

List. 1: Compartment trampoline pseudo code

to change the functions prototypes and implementations from *long* to pointers (*uintptr_t*, which could hold a pointer or an integer). Similarly, *prctl* and *clone* system calls.

Another important system call that needed to be changed is *mmap*. For MMU-less Linux, it is mainly being used to allocate memory; it does not use any virtual mappings as there is no paging. Unlike the previous system calls, *mmap* returns a pointer to an allocated memory. The return type needed to also be changed from *long* to a pointer type that could hold a bounded valid capability (if memory allocation is successful). We also mapped Linux's VM permissions passed to *mmap* (e. g. RWX) to CHERI permissions and disabled access to system registers for returned *mmap* capabilities. Such permission flags passed to *mmap* are ignored in MMU-less Linux.

Pointers: This is architecture independent code that is confusing integers with pointers and addresses. The code usually uses *long* as a type (or casts to it) instead of using proper types such as pointers (e. g. *char* * or *intptr_t*) when dealing with addresses and pointers (e. g. defining base and end addresses or performing pointer arithmetic) that get de-referenced later. This causes capabilities to lose their metadata (including tags), and thus, trigger security violations when accessed later. Example code is the memory management subsystem that allocates and frees pages and defines memory regions and blocks (with base addresses and sizes). Another example is data structures such as radix-tree, rbtree, and maple-tree where they perform casting operations to retrieve parents from nodes, or convert nodes to entries (and vice versa) and perform some masking on pointers. File handling code also needed to be changed a bit. Functions that take file descriptors as arguments and return a buffer or struct addresses needed to return capabilities instead of *long* integers.

Provenance: First, drivers that deal with MMIO devices need to create capabilities for each region. For MMU-less Linux, this was in the FDT code where it gets the base addresses of devices and their MMIO sizes from the Flattened Device Tree (FDT) nodes. We then create CHERI capabilities for those which are returned and held in their drivers as pointers.

Similarly, *ioremap* needed to be changed to return new valid capabilities. Second, we needed to create capabilities on the fly for some ELF loading code. The current ELF specification uses *long* types to hold addresses/pointers (e. g. *Elf64_Phdr.p_vaddr* that holds addresses of program headers and segments). We do not want to change the ELF specification itself in this occasion, so we had to create data capabilities for the code that is offsetting ELF segments while loading ELF binaries.

Misc: This includes some changes that are not part of the above categories. For instance, Linux sometimes uses reserved unused variables for padding within some structs. The number (or types) of such unused variables needed to change to meet the required alignment, offsets, or padding. We also needed to use our own versions of string functions such as *memcpy*, *memmove*, *memcmp* to preserve tags. Other string-related functions such as *strscpy* and *strncpy* (from user), and *strlen* did out-of-bounds reads as well for lengths shorter than the size of *long*, so this needed to be fixed to read byte-by-byte.

Compartmentalisation in the kernel is done on loadable modules. Each kernel module could be built as a compartment by providing a new compiler flag in order to reference all function and variable references GP-relative, instead of PC-relative, as discussed in the CompartOS paper [Al22b]. Furthermore, each module has its own capability table, separate from the kernel's. External symbols referenced within the modules that belong to the kernel or other external modules are created and added to the module's capability table in its external capability section (see Figure 3 and Listing 1). References to external symbols trigger a protection domain switch as shown in Figure 3. The LoC changes to support the CompartOS compartmentalisation model in the kernel are shown in Table 2. This also includes handling new CompartOS GP-relative ELF relocations.

Compartmentalisation in userspace is realised by the fact that each CHERI C ELF-FDPIC program has its own separate capability table; communication between different ELFs and processes is left to Linux IPC mechanisms, and is not linkage-based external capabilities (as in CompartOS). The compartmentalised userspace is shown as in Figure 4. The kernel's ELF-FDPIC loader sets up the process environment and memory. It allocates a single chunk of memory for each ELF image, including *uldso* and *Busybox*. It then transfers the program flow to userspace along with restricted number of capabilities passed as arguments. For *uldso*, the ELF-FDPIC loader hands over capabilities to the dynamic ELF section in order to perform the required ELF relocations. This includes relocations to normal ELF symbols and CHERI capabilities. It also gives *uldso* a capability to the ELF-FDPIC's load map structure, which itself includes subset capabilities to *Busybox's* loaded ELF segments. Once the relocation process is completed by *uldso*, the program flow transfers to *Busybox* with another set of restricted capabilities. The first capability is the root capability that *only* covers the entire *Busybox's* ELF loaded memory segments. The root capability gets used later (at process startup) to derive fine-grained capabilities for all global variables and functions and populates a confined capability table for *Busybox* and *uclibc-ng*; similar to the Linux's boot code. The second capability is the stack pointer which points to the process' arguments, ELF *auxv*, *envp*, etc., as expected by the POSIX environment. This

stack capability is set up by the ELF-FDPIC loader and gets passed to *uldso* which hands it over to *Busybox* as it is. Thus, each user process is only restricted by these two capabilities and cannot access anything else (e. g. other processes' memories).

4.2 Userspace

For the userspace, we use *Busybox* which is widely used in embedded MMU-less Linux to provide some embedded UNIX/POSIX utilities. We also use *uclibc-ng* as a C library for simplicity, which is also widely used in MMU-less Linux. Finally, a simple run-time linker was used called *uldso*.

4.2.1 uldso

uldso is a very simple and light-weight dynamic linker. It relies on Linux's ELF-FDPIC loader which is used for MMU-less systems. *uldso* assumes that the code is compiled position-independent with the *PIE* flag and the ELF-FDPIC loader has set up some registers and process segments at process startup. It then uses this information to fix up dynamic relocations. CHERIfying *uldso* enables both CHERI C user applications, and ELF-based compartmentalisation (user programs and libraries). The CHERI-LLVM toolchain emits a capability table and a relocation ELF section per ELF. Thus, this effectively sandboxes or compartmentalises each loaded ELF, and isolates it from one another; which gets us spatial memory isolation among ELFs, similar to what the MMU provides. Furthermore, supporting CHERI C ELFs gets us in-address-space spatial memory safety that MMU-based processes cannot have. The code changes to support CHERI are quite minimal. Startup RISC-V assembly code needed to deal with capability registers instead of integer registers, before jumping to the C's linker code. The linker simply needed to be taught about a new CHERI relocation section and its entries, and it then relocates each relocation entry with the run-time load address of the symbol's loaded segment in memory. This relocation section gets used later when the actual ELF bootstraps during the CHERI C initialisation process, which basically populates the capability table with valid capabilities.

4.2.2 uclibc-ng

The low-level part of this is quite similar to the Linux kernel. The capability table is populated at start-up, using the capability relocation info, fixed up by *uldso*. Atomics, context switching, *setjump/longjump*, also needed to deal with capability registers. There are also some low-level system calls parts in RISC-V assembly (e. g. *vfork* and *clone*) that needed to be modified to pass and return CHERI capability registers. The system call changes were minimal, and basically required to change the assumption and types of registers, arguments, and return values from *integers* and *longs* to capability registers (see

```
-long syscall(long sysnum, ...)
+uintptr_t syscall(long sysnum, ...)
 {
-       unsigned long arg1, arg2, arg3, arg4, arg5, arg6;
+       uintptr_t arg1, arg2, arg3, arg4, arg5, arg6;
        va_list arg;
        va_start (arg, sysnum);
-       arg1 = va_arg (arg, unsigned long);
-       arg2 = va_arg (arg, unsigned long);
-       arg3 = va_arg (arg, unsigned long);
-       arg4 = va_arg (arg, unsigned long);
-       arg5 = va_arg (arg, unsigned long);
-       arg6 = va_arg (arg, unsigned long);
+       arg1 = va_arg (arg, uintptr_t);
+       arg2 = va_arg (arg, uintptr_t);
+       arg3 = va_arg (arg, uintptr_t);
+       arg4 = va_arg (arg, uintptr_t);
+       arg5 = va_arg (arg, uintptr_t);
+       arg6 = va_arg (arg, uintptr_t);
```

List. 2: Git diff of the uclibc-ng common system call function

Listing 2). Minor changes to *malloc* were also required to enforce pointer-sized alignments and granularity. Furthermore, internal *malloc* code was using integer types for blocks and their arithmetic, that needed to be changed to capabilities as well.

The part that required most additions is libc's string functions. This includes *memcpy* family, and *str** functions. They were copied from Linux. *memcpy* and *memmove* needed to perform capability-aware operations to preserve tags. *str** functions were mostly doing out-of-bounds accesses, trying to aggressively optimise performance by reading word-sized chunks of memory. We used byte-based versions of those that are less optimised but do not perform out-of-bounds accesses.

4.3 Busybox

Busybox was quite cleanly written, meaning that almost no changes were required at all to build and run it in CHERI C mode. Minor changes to low-level assembly code were required to read capability variables instead of integers. Most importantly, we can use *insmod* command and friends to load kernel modules as CompartOS compartments.

5 Future Work

This work is still in progress and has been only evaluated on *QEMU* for functional correctness and applicability. We plan to keep enhancing it by cleaning up the code and enabling more

Linux features (e. g. networking, VirtIO, devices, etc). We will also aim to run on actual CHERI-RISC-V and/or Morello hardware in order to perform performance evaluation and run stock benchmarks, similar to *CheriFreeRTOS* [Al22b]. On the security side, we will aim to reproduce some of the Linux memory safety CVEs and see if CHERI could protect against them, in terms of integrity, confidentiality, and availability. As Linux in general is a demanding high-end system, evaluating scalability in terms of the number of user programs, libraries, and kernel modules would also be on our future work. We have not attempted to use CHERI sub-object bounds enforcement. This is an alternative compilation mode that automatically narrows capability bounds for pointers to individual structure members and arrays, restricting the possibility of intra-object memory corruption. Consider, for example, the C structure in Figure 5. In a pure-capability program, a pointer to `struct sensitive_data` produced by an allocator is mapped to a capability bounded to the size of the object (i. e. the *length* of the capability is `sizeof(struct sensitive_data)`). In the default CHERI compilation mode, pointers to sub-object members inherit the bounds of the object capability. This means that the capability for the `buffer` field has the same bounds as the parent structure. As a result, a buffer overflow of the `buffer` array is not detected as long as it remains within the bounds of the parent structure. This is clearly a limitation as intra-object memory safety is not provided.

```
struct sensitive_data {
  int some_value;
  char buffer[128];
  struct sensitive_data *next;
};

struct sentitive_data *p;
// Does not trigger a fault without sub-object bounds
p->buffer[130] = 'a';
// Triggers a fault with and without sub-object bounds
p->buffer[200] = 'a';
```

Fig. 5: Example of a structure that benefits from CHERI sub-object bounds enforcement. Note that, without sub-object bounds enforcement, buffer overflows (and underflows) via the `p->buffer` pointer are not prevented as long as they remain within the bounds of `struct sensitive_data`.

The sub-object bounds compilation mode provides superior security properties, however it exhibits larger friction for porting existing code bases [Ri20] due to some problematic C-language idioms. It should be noted that the CheriBSD pure-capability kernel already provides sub-object bounds enforcement, therefore there is reason to believe that the same level of support can be achieved in Linux. Previous experience with the CheriBSD pure-capability kernel has found that the majority of the portability issues arise from the use of `containerof()` patterns, where a pointer to the container structure is constructed from a pointer to a member. These kinds of patterns break in the presence of sub-object bounds, however CheriBSD has shown that it is feasible to annotate these problematic cases to selectively opt-out of sub-object bounds while maintaining the security benefits for most

of the code-base. Future work will evaluate the use of sub-object bounds to improve upon the base CHERI spatial memory safety guarantees.

Finally, we have not tried to support or evaluate temporal safety yet. Future work will significantly enhance security by applying some of the CHERI-based temporal safety techniques such as CherIvoke [Xi19] or CapRevoke [Fi20].

6 Related Work

ACES [Cl18] is an MPU-based software compartmentalisation approach that aims to automatically create compartments at build time by statically analysing the source code and an input security policy. It relies on off-the-shelf MPUs to enforce memory protection and instruments the final binary with MPU-based compartment switches. However, ACES only targets simple static systems, and would not scale to dynamic and high-end embedded systems like embedded Linux.

MINION [Ki18] is an MPU-based software security architecture for embedded systems that provides memory isolation between processes while optimising performance by avoiding performing frequent systems calls that are often associated with multi-privilege rings and MPUs. The paper argues that frequent system calls usually violate real-time system constraints and responsiveness. It shares the same limitations as MPU-based techniques as with ACES.

uVisor [AR] also uses Arm's MPUs and is similar to MINION as it can run an RTOS and other compartments in an unprivileged ring. However, it is different from MINION as it dynamically loads and creates compartments at runtime and is able to place different software entities in a compartment such as interrupt handlers or linkage-based modules, besides threads. Thus, it is not only a task-based compartmentalisation approach.

CHERI has been under research in UNIX-based environments with MMU, prototyped in the CheriBSD OS (a CHERI-enabled fork of the FreeBSD OS). CheriABI [Da19] is an application-level software compartmentalisation technique in CheriBSD. The main software application in CheriABI is C/C++ language pointer safety at the user level with a few modifications to the FreeBSD kernel. Two compilation modes are supported for CHERI: hybrid and pure-capability modes. In hybrid mode, pointers are integers as usual, and only those annotated with *__capability* keywords are protected by CHERI. CheriABI falls in the pure-capability category where user processes are compiled to have all pointers, system call arguments and allocated C objects (such as malloc and TLS) represented as CHERI capabilities. This significantly enhances spatial memory safety in UNIX while it is still being compatible with native UNIX processes that are not aware of CHERI. There is still ongoing research to have the FreeBSD kernel itself making full use of CHERI to compartmentalise the kernel components and enforce pointer safety. This is known as a pure-capability CheriBSD kernel. The CheriBSD kernel makes extensive use of CHERI memory safety

features, including spatial and referential memory safety, as well as sub-object bounds. In particular, new abstractions for virtual memory management are necessary to ensure the representability of mappings. Similarly, kernel allocators enforce representable bounds. The use of sub-object bounds shows promising results to protect from intra-object memory corruption.

CheriRTOS [Xi] is an early exploration of CHERI in small embedded systems. Unlike CheriABI (that targets UNIX-based systems), CheriRTOS targets microcontrollers to offer hardware protection using 64-bit compressed CHERI-MIPS capabilities. The paper shows how fine-grained memory protection, task isolation, secure heap management, and secure cross-domain transition can be implemented on CHERI. The authors also argue that the MPU, which is usually being used for memory protection in safety-critical embedded systems, is impractical as it does not meet the fine-grained memory protection requirements. Configuring the MPU takes a considerable number of cycles in kernel mode and is inefficient when it comes to the power consumption and die area.

CHERIoT [Am23] is targeting low-end IoT devices and aims to secure them using CHERI. It uses similar GP-relative addressing to CompartOS, however, CHERIoT is not designed for dynamic systems or high-end embedded systems such as embedded Linux.

Huawei's CheriLinux [cl], though completely separate, is the closest to our work, specifically only to our first contribution (CHERI C Linux). They target MMU-based server-class Linux, unlike our work here which targets MMU-less Linux. Furthermore, they only support CHERI C systems without software compartmentalisation at all, which we provide as three separate contributions.

7 Conclusion

We have shown how MMU-less Linux, one of the most advanced high-end operating systems, could be secured using two CHERI protection models: CHERI C and compartmentalisation. CHERI C works at the language level by protecting every pointer in the kernel and userspace. This provides complete spatial pointer safety. Compartmentalisation splits up a monolithic system such as the kernel and userspace into smaller and isolated logical compartments such as kernel modules in the kernel, and user applications and libraries in userspace. This reduces the effects of future unknown attacks.

We have demonstrated the effort it took in order to secure the upstream RISC-V MMU-less Linux, *Busybox* and *uclibc-ng* sysetms. The LoC changes were quite minimal for such large code bases written in millions LoC, which emphasises the practicality and compatibility of securing low-level software using CHERI with minimum effort. This completely relies on CHERI as the sole hardware protection mechanism. We further show that the CompartOS model [Al22b, Al22a] is applicable to large high-end embedded systems such as Linux.

Bibliography

[Al22a] Almatary, Hesham: CHERI compartmentalisation for embedded systems. Technical Report UCAM-CL-TR-976, University of Cambridge, Computer Laboratory, November 2022.

[Al22b] Almatary, Hesham; Dodson, Michael; Clarke, Jessica; Rugg, Peter; Gomes, Ivan; Podhradsky, Michal; Neumann, Peter G; Moore, Simon W; Watson, Robert NM: CompartOS: CHERI Compartmentalization for Embedded Systems. arXiv preprint arXiv:2206.02852, 2022.

[Al23a] Almatary, Hesham: , Busybox. https://github.com/heshamelmatary/busybox, 2023.

[Al23b] Almatary, Hesham: , CheriLinux. https://gitlab.com/hmka/cherilinux, 2023.

[Al23c] Almatary, Hesham: , uClibc-ng. https://github.com/heshamelmatary/uclibc-ng, 2023.

[Al23d] Almatary, Hesham: , uldso. https://github.com/heshamelmatary/uldso, 2023.

[Am23] Amar, Saar; Chen, Tony; Chisnall, David; Domke, Felix; Filardo, Nathaniel; Liu, Kunyan; Norton-Wright, Robert; Tao, Yucong; Watson, Robert NM; Xia, Hongyan: CHERIoT: Rethinking security for low-cost embedded systems. Technical report, Technical Report MSR-TR-2023-6. Microsoft. https://www. microsoft. com/en-us . . . , 2023.

[AR] ARMmbed: , The Arm Mbed uVisor. https://github.com/ARMmbed/uvisor.

[Bu] BusyBox: , Busybox. https://busybox.net.

[Ci19] Cimpanu, Catalin: , Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/, Feb 2019.

[cl] cheri linux, GitHub: , GitHub - cheri-linux. https://github.com/cheri-linux.

[Cl18] Clements, Abraham A; Almakhdhub, Naif Saleh; Bagchi, Saurabh; Payer, Mathias: {ACES}: Automatic compartments for embedded systems. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 65–82, 2018.

[Co98] Cowan, Crispan; Pu, Calton; Maier, Dave; Walpole, Jonathan; Bakke, Peat; Beattie, Steve; Grier, Aaron; Wagle, Perry; Zhang, Qian; Hinton, Heather: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX security symposium. volume 98. San Antonio, TX, pp. 63–78, 1998.

[Da19] Davis, Brooks; Watson, Robert N M; Richardson, Alexander; Neumann, Peter G; Moore, Simon W; Baldwin, John; Chisnall, David; Clarke, Jessica; Filardo, Nathaniel Wesley; Gudka, Khilan; Others: CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 379–393, 2019.

[DI] DDC-I, Inc.: , Deos: Safety Critical RTOS for Avionics Applications requiring DO178C/ED-12C DAL A verification. https://web.archive.org/save/https://www.ddci.com/products_deos_do_178c_arinc_653.

[Fi20] Filardo, Nathaniel Wesley; Gutstein, Brett F; Woodruff, Jonathan; Ainsworth, Sam; Paul-Trifu, Lucian; Davis, Brooks; Xia, Hongyan; Napierala, Edward Tomasz; Richardson, Alexander; Baldwin, John et al.: Cornucopia: Temporal safety for CHERI heaps. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 608–625, 2020.

[Fra] FreeRTOS: , AWS Reference Integrations. https://www.freertos.org/aws-reference-integrations.html.

[Frb] FreeRTOS: , FreeRTOS - Stack Usage and Stack Overflow Checking. https://www.freertos.org/Stacks-and-stack-overflow-checking.html.

[Gu15] Gudka, Khilan; Watson, Robert NM; Anderson, Jonathan; Chisnall, David; Davis, Brooks; Laurie, Ben; Marinos, Ilias; Neumann, Peter G; Richardson, Alex: Clean application compartmentalization with soaap. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1016–1031, 2015.

[Ha88] Hardy, Norm: The Confused Deputy:(or why capabilities might have been invented). ACM SIGOPS Operating Systems Review, 22(4):36–38, 1988.

[Ka87] Karger, Paul A: Limiting the damage potential of discretionary Trojan horses. In: 1987 IEEE Symposium on Security and Privacy. IEEE, pp. 32–32, 1987.

[Ki03] Kilpatrick, Douglas: Privman: A Library for Partitioning Applications. In: USENIX Annual Technical Conference, FREENIX Track. pp. 273–284, 2003.

[Ki18] Kim, Chung Hwan; Kim, Taegyu; Choi, Hongjun; Gu, Zhongshu; Lee, Byoungyoung; Zhang, Xiangyu; Xu, Dongyan: Securing Real-Time Microcontroller Systems through Customized Memory View Switching. (February), 2018.

[Kl09] Klein, Gerwin; Elphinstone, Kevin; Heiser, Gernot; Andronick, June; Cock, David; Derrin, Philip; Elkaduwe, Dhammika; Engelhardt, Kai; Kolanski, Rafal; Norrish, Michael; Others: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220, 2009.

[Ko18] Kocher, Paul; Genkin, Daniel; Gruss, Daniel; Haas, Werner; Hamburg, Mike; Lipp, Moritz; Mangard, Stefan; Prescher, Thomas; Schwarz, Michael; Yarom, Yuval: Spectre attacks: Exploiting speculative execution. arXiv preprint arXiv:1801.01203, 2018.

[L4] L4Re: , L4 Runtime Environment (L4Re). https://l4re.org/doc/.

[La73] Lampson, Butler W: A note on the confinement problem. Communications of the ACM, 16(10):613–615, 1973.

[Le84] Levy, Henry M: Capability-based computer systems. Digital Press, 1984.

[Le17] Levy, Amit; Campbell, Bradford; Ghena, Branden; Giffin, Daniel B; Pannuto, Pat; Dutta, Prabal; Levis, Philip: Multiprogramming a 64kb computer safely and efficiently. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 234–251, 2017.

[Li95] Liedtke, Jochen: On micro-kernel construction, volume 29. ACM, 1995.

[NA] NASA: , NASA: Magnetosphere Multiscale (MMS) Mission. https://mms.gsfc.nasa.gov/.

[On96] One, Aleph: Smashing the stack for fun and profit. Phrack magazine, 7(49):14–16, 1996.

[PFH03] Provos, Niels; Friedl, Markus; Honeyman, Peter: Preventing Privilege Escalation. In: USENIX Security Symposium. 2003.

[PMB15] Papp, Dorottya; Ma, Zhendong; Buttyan, Levente: Embedded systems security: Threats, vulnerabilities, and attack taxonomy. 2015 13th Annual Conference on Privacy, Security and Trust, PST 2015, pp. 145–152, 2015.

[PW08] Parameswaran, Sri; Wolf, Tilman: Embedded systems security—an overview. Design Automation for Embedded Systems, 12(3):173–183, 2008.

[Ri02] Richarte, Gerardo et al.: Four different tricks to bypass stackshield and stackguard protection. World Wide Web, 1, 2002.

[Ri20] Richardson, Alexander: Complete spatial safety for C and C++ using CHERI capabilities. Technical Report UCAM-CL-TR-949, University of Cambridge, Computer Laboratory, 2020.

[RTa] RTEMS: , RTEMS - Stack Bounds Checker. https://docs.rtems.org/branches/master/c-user/stack_bounds_checker.html.

[RTb] RTEMS: , RTEMS on board NASA MMS scheduled for launch this Thursday! https://www.rtems.org/node/118.

[sP] seL4 Project: , What is Proved and What is Assumed | seL4. https://web.archive.org/web/20220720085604/https://sel4.systems/Info/FAQ/proof.pml.

[SS75] Saltzer, Jerome H; Schroeder, Michael D: The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[SSF99] Shapiro, Jonathan S; Smith, Jonathan M; Farber, David J: EROS: a fast capability system. In: Proceedings of the seventeenth ACM symposium on Operating systems principles. pp. 170–185, 1999.

[Sz13] Szekeres, Laszlo; Payer, Mathias; Wei, Tao; Song, Dawn: Sok: Eternal war in memory. In: Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, pp. 48–62, 2013.

[una] uClibc ng: , uClibc-ng - Embedded C library. https://uclibc-ng.org.

[Unb] Ungerer, Greg: , GitHub - uldso. https://github.com/gregungerer/uldso.

[Wa10] Watson, Robert N M; Anderson, Jonathan; Laurie, Ben; Kennaway, Kris: Capsicum: Practical Capabilities for UNIX. USENIX Security Symposium, 46:2, 2010.

[Wa15] Watson, Robert N M; Woodruff, Jonathan; Neumann, Peter G; Moore, Simon W; Anderson, Jonathan; Chisnall, David; Dave, Nirav; Davis, Brooks; Gudka, Khilan; Laurie, Ben; Others: Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp. 20–37, 2015.

[Wi21a] Wikipedia contributors: , BeagleBoard — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=BeagleBoard&oldid=1053954096, 2021. [Online; accessed 16-February-2022].

[Wi21b] Wikipedia contributors: , NXP ColdFire — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=NXP_ColdFire&oldid=1031125403, 2021. [Online; accessed 16-February-2022].

[Wi21c] Wikipedia contributors: , QorIQ — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=QorIQ&oldid=1057508237, 2021. [Online; accessed 31-January-2022].

[Wi22a] Wikipedia contributors: , Buffer overflow protection — Wikipedia, The Free Encyclopedia, 2022. [Online; accessed 29-January-2022].

[Wi22b] Wikipedia contributors: , Raspberry Pi — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=1072126107, 2022. [Online; accessed 16-February-2022].

[WRS] Wind River Systems, Inc: , VXWORKS CERT PLATFORM PRODUCT OVERVIEW. https://web.archive.org/web/20211124075819/https://www.windriver.com/resources/product-overviews/vxworks-cert-product-overview.

[Wu74] Wulf, William; Cohen, Ellis; Corwin, William; Jones, Anita; Levin, Roy; Pierson, Charles; Pollack, Fred: Hydra: The kernel of a multiprocessor operating system. Communications of the ACM, 17(6):337–345, 1974.

[Xi] Xia, Hongyan; Woodruff, Jonathan; Barral, Hadrien; Esswood, Lawrence; Joannou, Alexandre; Kovacsics, Robert; Chisnall, David; Roe, Michael; Davis, Brooks; Napierala, Edward; Baldwin, John; Gudka, Khilan; Neumann, Peter G; Richardson, Alexander; Moore, Simon W; Watson, Robert N M: CheriRTOS: A Capability Model for Embedded Devices. In: Proceedings of the 36th IEEE International Conference on Computer Design (ICCD '18). IEEE, pp. 92–99.

[Xi19] Xia, Hongyan; Woodruff, Jonathan; Ainsworth, Sam; Filardo, Nathaniel W; Roe, Michael; Richardson, Alexander; Rugg, Peter; Neumann, Peter G; Moore, Simon W; Watson, Robert NM et al.: Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 545–557, 2019.

[Zh19] Zhou, Wei; Guan, Le; Liu, Peng; Zhang, Yuqing: Good motive but bad design: Why ARM MPU has become an outcast in embedded systems. arXiv preprint arXiv:1908.03638, 2019.

A Preliminary Survey of the State of the Art in Simulation-Based Development and Certification to Support Digital Aircraft Design Research

Malte Christian Struck[1], Alexander Weinert[2], Andreas Schreiber[2], Michael Felderer[2,3]

Abstract: Many safety critical domains require certification of a product before it can be released to the market. On the one hand, simulations and digital methods allow for cheaper and faster assurance of properties. On the other hand, the new and different methodology implies completely new requirements. We provide a general overview of the field and focus on naming, use cases, stakeholders, and quality criteria. We also highlight the needs of simulation users for research. We show that the naming in the different domains for virtual certification is widely spread, but the research needs aim into the same direction.

Keywords: Virtual Certification; Certification by Analysis; Simulation; Development; Certification; CbA; Numerical method; Simulation Based Development

1 Introduction

Certification is a critical step in the development of novel aircraft. The current certification process requires the iterative construction of physical prototypes of the aircraft, which then undergoes numerous tests to ensure its airworthiness. Each iteration is a waterfall process comprised of design, prototype construction and tests. Only after a successful test a certification can be obtained. If the test fails the whole iteration has to be redone which lengthens the time from initial design to final market-ready product.

A major research trend in aerospace engineering is the digital transformation of the aircraft development process, which includes the certification of the aircraft [Ma21; Ma22]. The authors are working at the Virtual Product House (VPH), an initiative of German Aerospace Center (DLR) which aims to develop novel methods and processes for virtual aircraft

[1] Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Softwaretechnologie, Cornelius-Edzard-Straße 15, 28199 Bremen, Deutschland malte.struck@dlr.de
[2] Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Softwaretechnologie, Linder Höhe, 51147 Köln, Deutschland {alexander.weinert,andreas.schreiber,michael.felderer}@dlr.de
[3] Universität zu Köln, Department Mathematik/Informatik, Albertus-Magnus-Platz, 50931 Köln, Deutschland

doi:10.18420/sw2024-ws_07

development. This includes the development of higher fidelity simulation methods, the development of composite simulations that simulate the aircraft manufacturing process, and the development of methods for long-term storage of simulated data. During this work we have observed that a number of requirements towards this newly developed methods are independent of the aviation domain, e. g., the requirement to record the provenance of simulation data, or the necessity to store these data over the life-time of the product. Similarly, some of the newly developed methods are also independent of the aviation domain. This includes, for example, the visualization of the execution of composite simulations [MSW23], or the integration of such composite simulations with a data management system [Dr22].

Based on this observation, we suspect that similar developments are currently taking place in other regulated domains where safety-critical products are certified before entering the market. These include, for example, the space industry, the automotive industry, and the medical industry. Our initial goal is to determine the current state of the art in digital transformation of the product development and certification process across multiple domains. This is complicated by the fact that the digital transformation of the engineering process is still a nascent field, with very few cross-domain standards developed to date.

Even within aviation, work related to digital transformation is sometimes discussed under the term "virtual development and certification" and sometimes under the term "certification by analysis." The latter term implicitly includes simulation-based aircraft development.

Our goal is to identify research groups, existing results, and nomenclature across domains. This will facilitate the reuse of results and methods. To this end, we plan several activities:

- First, we identify researchers working towards virtual certification and ask them to answer a few standardized questions regarding the state of the art of virtual certification in their respective domain.
- Second, we use the results of the first step to identify key contributors to the topic of virtual certification and conduct in-depth interviews with these contributors.
- Third, we use the results of the in-depth interviews to conduct a structured literature review to gain a broader understanding of current research trends in different domains.

In this work, we present preliminary results of the first stage of this research plan. In Section 2, we give an overview over the digital transformation of the aircraft development process as it is understood at VPH. There, we moreover identify open questions in virtual certification for aircraft, which we believe to be important in other domains as well. Afterwards, we describe our research plan for the questionnaire in greater depth in Section 3. Subsequently, we give an overview over preliminary results of the initial phase in Section 4. Finally, we discuss our next steps in Section 5.

2 Virtual Certification at VPH

The VPH [Ge] is a research collaboration comprised of research institutes as well as aviation industry partners, and members of the European Union Aviation Safety Agency (EASA).

It focuses on the enhancement of digital methods for aircraft development, testing, and certification. The long-term goal of VPH is to develop methods that allow for a completely virtual certification of aircraft and aircraft parts. The intended development and certification process is mapped to a digital tool-chain containing abilities from various aviation-related disciplines.

In VPH the whole process is seen as a digital end-to-end process, starting with requirements over virtual manufacturing to a virtually tested and certified product; Rädel et al. [Rä21] and Lange et al. [La21] give an in-depth description of this process. This process is realized using the workflow engine Remote Component Environment (RCE), which allows engineers to design and execute highly distributed tool-chains. For RCE, refer to Boden et al. [Bo21] for a technical description and Flink et al. [Fl22] for a user-focused introduction. The VPH tool-chain comprises several discipline-specific capabilities, such as virtual methods for testing an actuation system due to Hollmann et al. [Ho22], aerodynamic simulations due to Zakrzewski et al. [ZLH22], or automated aircraft model generation and sizing due to Führer et al. [Fü16].

The VPH follows the long term goal of virtual certification. Hence, a necessary subgoal is the credibility of our data and its processing within the VPH tool-chain. Thus we research how provenance data, i. e., information about the origin of data, has to be created and stored during and after the execution of the tool-chain [MSW23]. A data-driven view on this topic is given by Dressel et al. [DD21] and Dressel et al. [Dr22].

To be able to use simulation data for development and certification, engineers have to approve that their results map to a physical test, i. e., they have to ascertain the validity of the simulated result. Thus we aim to determine whether domains other than aviation use quality criteria or even a metric for validating their simulation results.

While working on the topic of data provenance at VPH, we observed that many data-related research questions are not specifically related to aviation. Therefore, these research questions related to data provenance are likely to either be addressed or to be of use in other domains. We already showed in previous work how provenance data is generated and stored in the current VPH tool-chain [DD21; Dr22]. Our current solution, however, has not yet been shown to scale to larger tool-chains. Furthermore we aim to investigate methods that allow data to be stored together with its provenance for the lifetime of the developed product. Since the lifetime of an individual product line in aviation often reaches several decades, this is decidedly non-trivial.

No model will ever exactly reflect the real-world situation. For certification, we need to quantify the difference between the model's prediction and the real-world phenomena. Thus,

quantifying the uncertainty of model results and the models' robustness against minor deviations in the input data become an important topic. We are interested in how other domains deal with this matter.

3 Methodology

To gain initial insight into the processes used in domains that are supposed to use simulations for development and certification (not only limited to aviation), we created a questionnaire asking respondents to give an overview over the use of simulations for product development in their respective domain. As this is performed in the initial stage of our research we only asked high-level questions to gain a broad insight into this field. We show that questionnaire in Table 1. We sent out a questionnaire to ~30 experts from the domains automotive, aviation, space, biotechnology and maritime and received the answers of ten researchers in the automotive domain. Moreover, we answered the questionnaire ourselves to obtain comparable answers from the aviation domain. We show detailed information about the participants' demographics in Table 2.

Q1	What terms are used in your domain to address simulation-based development or certification? Mit welchen Begriffen wird in Ihrer Domäne die simulationsbasierte Entwicklung oder Zertifizierung adressiert?
Q2	How are simulations used in your domain for product development or product certification? You may name a representative process. Wie werden in Ihrer Domäne Simulationen zur Produktentwicklung oder -zertifizierung eingesetzt? Nennen Sie gern einen repräsentativen Prozess.
Q3	Where do you see the largest need for research and development in the area of simulation-supported development and certification? Wo sehen Sie den größten Forschungs- und Entwicklungsbedarf im Bereich der simulationsunterstützten Entwicklung und Zertifizierung?
Q4	By which institution (certification agency, standardization body, internal regulations, . . .) are those requirements defined whose satisfaction is indicated via the simulations? Durch welche Instanz (Zulassungsbehörde, Standardisierungsgremium, interne Vorgaben, . . .) werden die Requirements vorgegeben, deren Erfüllung die Simulationen anzeigen?
Q5	Which metrics / criteria of quality are used for simulation-based verification methods? Welche Gütekriterien / Qualitätsmetriken werden für simulationsunterstützte Nachweismethoden verwendet?
Q6	Which stakeholders are involved in simulation-based development or certification? Welche Stakeholder sind an der simulationsunterstützten Entwicklung oder Zertifizierung beteiligt?
Q7	Please state your current job title or your domain of work, respectively, as well as the duration of your experience in this job and associated training. Bitte nennen Sie Ihren aktuellen Beruf bzw. Ihre Branche sowie die Dauer Ihrer Berufs- und Ausbildungserfahrung in Ihrem Bereich.

Tab. 1: The questions posed in our initial E-Mail survey (original German formulation shown in small print).

Domain	Job Title	Experience
Aviation	Scientific Researcher	1 year
Automotive	Scientific Researcher	10 years
Automotive	Team of Scientific Researchers (9 Members)	3 to 20 years

Tab. 2: The demographics of the respondents to our initial survey.

In this initial step we aimed to keep the barrier to entry for the researchers as low as possible. Hence, we only posed a few brief questions. We moreover chose to conduct the survey via email instead of a more sophisticated survey tool, again to keep the barrier to entry as low as possible.

This questionnaire contained seven open questions, one of which only queried for demographic information. The other six questions are mapping to our research questions. Since all researchers identified spoke German fluently, we posed the questions in German.

We subsequently consolidated the answers from all participants and used them to evaluate our initial hypothesis and to inform our future research directions. We discuss these results in the following section.

4 Initial Results

We show the results of our preliminary and non-representative survey in Table 3. Where original answers were given in German, we have translated them into English.

We see that the answer to Q1 (Terms Used) directly supports our assumption that the digital transformation of the product development is discussed both in aviation as well as in the automotive domain. In fact, even though we only gathered data from two domains we obtained 19 different terms that are used in those domains to refer to simulation-based development or certification. Hence, we conclude that even within individual domains, no generally-agreed upon nomenclature has been established. Nevertheless we classified the answers to Q1 into the following non-distinct categories: a) In-the-Loop Testing b) Digital / Virtual Twin c) Simulations d) User-centric approaches e) Scenario-focused analyses f) Others

Moreover, we observe significant overlap between the individual terms given, both within and across the domains. In the automotive domain, e. g., both “User Study in Virtual Reality” and “User Study in the Driving Simulator” were named by researchers as types of user studies used in digital development processes. The common term “user study”, in contrast, refers to user studies without relevance to virtual development and is unfit as a characterizing term for the field. Similarly, the terms “Virtual Testing” and “Scenario-based Verification / Testing” were given by researchers from the aviation domain and the automotive domain, respectively.

Q1 (Terms Used)	User Study in Virtual Reality; User Study in the Driving Simulator; Safeguarding in the Digital Twin; Virtual Twin; Verification, Validation, Certification; Scenario-based Verification / Validation; Scenario-based Safety Assurance; Safeguarding / Testing / Engineering; xIL testing; Software-in-the-Loop (SiL) Testing; Hardware-in-the-Loop (HiL) Testing; Rare Event Simulation; Scenario-based Testing with the Aid of Driving Simulations; Traffic Planning and Optimization using Traffic Flow Simulations; Simulation-based virtual Integration Tests; Virtual Development; Virtual Certification; Virtual Testing; Certification by Analysis (CbA)
Q2 (Use Cases)	Feasibility Studies; Virtual Tests; User Studies; Recreation of equal Test Situations; Early error detection; Certification; Proof-of-concept in logistics; Virtual Integration Tests; Simulation of Surrounding Area; Demonstration of novel Interaction Concepts;
Q3 (Research Needs)	Validity of Simulations; Uncertainty Quantification; Robust Simulations
Q4 (Requirement Givers)	EASA; United Nations Economic Commission for Europe (UNECE); Regulation of the European Union; Users; Municipalities; Traffic Planners
Q5 (Quality Metrics)	Similarity to Validation Tests; Sharpness of the confidence estimate for the residual risk; Acceptance Criteria (ALARP, MEM,. . .); Criticality Metrics for Automated Driving
Q6 (Stakeholders)	OEMs; Tier 1 supplier; Approval authority; Vendors for Software Development Tools; Standardization Committees; Ministries (e. g. Transport Ministry); Urban Planners; Automotive Engineers; Research Institutes; Aircraft manufacturers

Tab. 3: Results from our initial survey

These observations show that we are unlikely to determine a fixed set of search terms or categories for a literature survey that allows us to obtain a comprehensive list of published works in this field across domains. Instead, we will most likely have to perform individual literature reviews per domain and consolidate the results to obtain a cross-domain review of the state of the art.

The answers to Q2 (Use Cases) show that the researchers expect various benefits from the digital transformation of the product development, ranging from earlier simplified demonstration of novel concepts, over error detection, to simplified certification. This is relatively unsurprising, as new developments are often met with major expectations of their usefulness once fully developed and deployed. These expectations often remain unmet during continued development [Ga].

Researchers' expectations contrast with their perceived requirement for future research, which we queried in Q3 (Research Needs). They identified the validity of simulations,

uncertainty quantification, and the robustness of simulations as the most important future research fields for the digital transformation of product development. Researchers identified not only a gap between the results of simulations and the results of real-world tests when using current methods, but also different behavior of human test subjects when working with physical objects and when working with their simulated counterparts. Hence, we believe that future research in this field should not be limited to engineering disciplines with the aim to improve the fidelity of simulations, but that it should also include psychological research into the interaction of humans with simulations. We believe that this human-focused research is essential to retain the validity of product evaluations during the digital transformation.

Clearly, the digital transformation of the product development process involves the development, deployment and use of bespoke software. Typically, when developing software, developers are advised to focus on the requirements of the end-user to obtain an optimal software product [SC17]. The responses to Q4 (Requirement Givers) and Q6 (Stakeholders) show, however, that for the digital transformation of product development a bespoke design process may be required, as there exist numerous stakeholders and requirement providers that are not end users of the developed process. Moreover, the number of stakeholders already appears to be too large to involve all of them in an agile development process. In our opinion, this observation should inform future efforts towards digital transformation of product development. Research projects should consciously work towards involving a representative subset of stakeholders, which may not coincide with the direct users of the research results.

The answers to Q5 (Quality Metrics) show that there is a lack of a criteria to decide whether a simulation is sufficient. We see that researchers conduct physical tests to validate the simulation results. They also mention the “sharpness of the confidence estimate” as a quality indicator. The questionnaire participants also mention acceptance criteria. The acceptance criteria mentioned, however, have the aim of determining acceptable risk. In our case, this amounts to deciding whether the risk of a potentially inadequate simulation is acceptable. This does not include judgements about the quality of the simulation. The same applies to criticality metrics as they give information about the consequences of a wrong quality estimation.

5 Conclusion and Future Work

In this work we have presented a research plan for investigating the state of the art in the digital transformation of the product development and product certification process in the aviation domain and the automotive domain. We moreover have presented hypotheses based on the results of an initial survey across researchers from the aviation domain as well as the automotive domain. Finally, we have gathered and presented initial data via this preliminary survey that allows us to conduct an in-depth elicitation of the state of the art of virtual certification across multiple domains. The results of our survey support the hypotheses set out in our research assumptions that a) the same research goals are pursued

in different domains, and that b) researchers in different domains use different nomenclature to refer to identical or very similar procedures and aims. In particular, we found that the automotive domain faces the same challenges with regards to the fidelity of simulations and their validation as the aviation domain. Moreover, researchers in the automotive domain already consider the challenge of different behavior of human test subjects in virtual and physical environments, which has not yet been considered at VPH. We moreover found that aviation researchers and automotive researchers identified virtually identical research needs: The topics of simulation validity, uncertainty quantification, and robustness of simulations with respect to minor disturbances in the input data appear to be of equal importance to researchers in both domains. Finally, we found that the surveyed researchers did not explicitly mention the need for trustworthiness of simulated data or its long-term storage. In previous work, in contrast, we found both topics to be highly relevant to certifying agencies in aviation and assume that they will be relevant in other regulated domains as well.

We believe that our results indicate that research towards simulation-based development and certification for digital aircraft design can immensely benefit from increased cooperation with other domains. One example of a topic where aviation research can benefit is the human behavior in simulated and physical environments. While this topic is already being researched in the automotive domain, it is not yet under investigation in aircraft development. We believe, however, that this topic will eventually become relevant as a building block for the digital end-to-end process developed at VPH (Section 2).

In future work, we aim to widen our research questions to other topics like software timing performance or human interaction to systematically investigate the state of the art in the digital transformation of product development and certification. In contrast to our initial survey, we aim to address additional safety-critical domains in which products are regulated and subject to certification. This includes, e. g., the space domain, the development of medical hardware, and the ship-building industry. For these domains, we will elicit the state of the art in virtual product development and certification via a) conducting in-depth interviews with selected researchers working in regulated domains and via b) performing a systematic literature review of existing work.

When conducting the in-depth interviews our aim is two-fold: First, we aim to determine the nomenclature commonly used in the respective domains when referring to virtual certification. As our initial survey shows, relevant topics are addressed under different names in different domains. By determining relevant nomenclature prior to the literature review, we can ensure that we include all relevant information in the review. Furthermore we can gain deeper insights regarding our research questions, e. g. the research needs, by interactively enquiring additional information. Secondly, we aim to identify the primary publication venues for the digital transformation of product development and certification in the respective domains via the interviews. Our experience from aviation shows that relevant literature is often published as gray literature, i. e., as technical reports [Ma21] or as memoranda issued by certification authorities [EA20]. Hence, determining relevant publication channels in the

respective domains will again allow us to ensure to comprehensively determine the state of the art in these domains.

Acknowledgments We thank the anonymous reviewers for their valuable comments, which greatly improved this manuscript. We moreover thank the interviewed researchers for their willingness to participate in the survey and for fruitful discussions. This work has partially been funded by the European Union as part of the SMR ACAP project (Grant Agreement ID 101101955).

References

[Bo21] Boden, B.; Flink, J.; Först, N.; Mischke, R.; Schaffert, K.; Weinert, A.; Wohlan, A.; Schreiber, A.: RCE: An Integration Environment for Engineering and Science. SoftwareX 15/, p. 100759, 2021.

[DD21] Dressel, F.; Doko, A.: Common Source & Provenance at Virtual Product House. In: DLRK 2021. DGLR, 2021.

[Dr22] Dressel, F.; Rädel, M.; Weinert, A.; Struck, M. C.; Haase, T.; Otten, M.: Common Source & Provenance at Virtual Product House: Integration with a Data Management System. In: DLRK 2022. DGLR, 2022.

[EA20] EASA: Notification of a Proposal to issue a Certification Memorandum. Modelling & Simulation – CS-25 Structural Certification Specifications./, Accessed on 23th October 2023, 2020.

[Fl22] Flink, J.; Mischke, R.; Schaffert, K.; Schneider, D.; Weinert, A.: Orchestrating Tool Chains for Model-based Systems Engineering with RCE. In: IEEE Aerospace Conference (AERO). 2022.

[Fü16] Führer, T.; Willberg, C.; Freund, S.; Heinecke, F.: Automated model generation and sizing of aircraft structures. Aircraft Engineering and Aerospace Technology 88/2, pp. 268–276, 2016.

[Ga] Gartner: Gartner Hype Cycle, Available at `https://www.gartner.com/en/research/methodologies/gartner-hype-cycle`, Accessed October 26, 2023.

[Ge] German Aerospace Center (DLR): Virtual Product House - Integration and Test Centre for Virtual Certification, `https://www.dlr.de/en/research-and-transfer/aeronautics/projects-guiding-concepts-for-dlr-aeronautics-research/the-virtual-product/virtual-product-house-integration-and-test-centre-for-virtual-certification`, Accessed October 27, 2023.

[Ho22] Hollmann, R. W.; Schäfer, A.; Bertram, O.; Rädel, M.: Virtual testing of multifunctional moveable actuation systems. en, CEAS Aeronautical Journal 13/4, pp. 979–988, Oct. 2022.

[La21] Lange, F.; Zakrzweski, A.; Rädel, M.; Hollmann, R.; Risse, K.: Digital Multi-Disciplinary Design Process for Moveables at Virtual Product House. In: DLRK 2021. 2021.

[Ma21] Mauery, T.; Alonso, J.; Cary, A.; Lee, V.; Malecki, R.; Mavriplis, D.; Medic, G.; Schaefer, J.; Slotnick, J.: A Guide for Aircraft Certification by Analysis, tech. rep., Available at https://ntrs.nasa.gov/citations/20210015404, NASA, 2021.

[Ma22] Mauery, T.; Alonso, J. J.; Cary, A. W.; Lee, V.; Malecki, R.; Mavriplis, D. J.; Medic, G.; Schaefer, J.; Slotnick, J. P.: A 20-year Vision for Flight and Engine Certification by Analysis. In: AIAA SCITECH 2022 Forum. American Institute of Aeronautics and Astronautics, Jan. 2022.

[MSW23] Meinecke, A.; Struck, M. C.; Weinert, A.: Visualizing RCE Workflow Executions via W3C Provenance. In: 2023 IEEE Aerospace Conference (AERO). 2023.

[Rä21] Rädel, M.; Lange, F.; Hollmann, R. W.; Risse, K.; Wille, T.: DLR Virtual Product House (VPH): Digital end-to-end development process for multifunctional moveables, https://elib.dlr.de/146402/, unpublished, 2021.

[SC17] Still, B.; Crane, K.: Fundamentals of User-centered Design, A practical approach. CRC Press, Boca Raton, 2017, ISBN: 9781498764445.

[ZLH22] Zakrzewski, A.; Lange, F.; Hollmann, R.: Multi-fidelity Aerodynamic Design Process for Moveables at DLR Virtual Product House. In: AIAA AVIATION 2022 Forum. AIAA, 2022, eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2022-3938.

Towards COTS component synchronization for low SWaP-C flight control systems

Franz Sax [1], Florian Holzapfel[2]

Abstract: The rise of innovative and novel fly-by-wire air vehicles like e-VTOLs for Advanced Air Mobility demands flight control systems whose components are low size, weight, power and cost (SWaP-C), but nevertheless offer high performance. One approach towards this mismatch is to use COTS components from e.g. the automotive sector and use their extensive features to enhance performance in a given system architecture. This paper describes one method of minimizing the latency in the communication between two COTS components by using an easily realizable algorithm with minimal memory, code and computation requirements for relative synchronization of the execution cycles of the components. A description of the resulting control problem, as well as simulation results from a dedicated MATLAB simulation environment are given. Those are then compared with an implementation on a representative set of devices from the EPUCOR flight control system.

Keywords: Synchronization; time-triggered; event-triggered; latencies; real time system; low level software; timing; drift

1 Introduction

Typical Commercial off-the-shelf (COTS) - based digital flight control systems for prototypical novel air vehicles are asynchronous. This means that the different components periodically perform their tasks - but in a way that the execution period of one device does not have a well defined relation to the execution period of other devices in the system architecture.

With the following small example, the situation shall be explained: In the flight control system for a 600kg unmanned helicopter [B.22], the main Inertial Navigation System (INS) sensor is directly connected to the primary Flight Control Computer (FCC) using a single A429 connection. This sensor's data is the most important one for the essential task of stabilizing the aircraft in flight. With a period of 20ms, the sensor transmits data packets with the newest measured data. The COTS flight control computer also performs all of its actions in a 20ms-loop: collecting and parsing input data, executing the controller algorithm and finally packing and sending the computed data and commands. The execution time of

[1] TUM-FSD, Boltzmannstr. 15, D-85748 Garching, Germany, franz.sax@tum.de
[2] TUM-FSD, Boltzmannstr. 15, D-85748 Garching, Germany, florian.holzapfel@tum.de

doi:10.18420/sw2024-ws_08

the control algorithm and I/O handling is such, that there is a certain idle time after sending out the commands. In this time, other lower priority tasks are executed by the implemented real time operating system (RTOS).

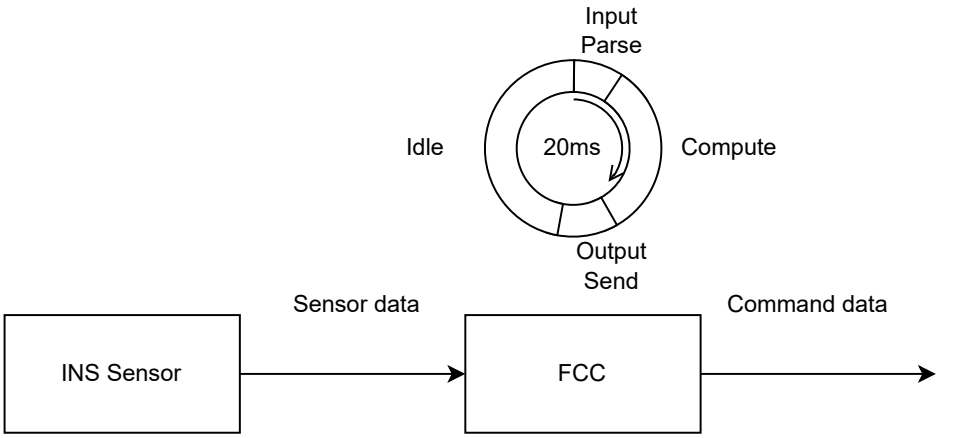


Fig. 1: Overview over the relevant part of an exemplary system architecture. The sensor sends new data every 20ms and the FCC also restart its execution cycle every 20ms

However, the 20ms-cycles of the two devices do - after turning on the system - initialize in a random offset relation and further will drift relative to each other, coming from the fact that the internal timing sources, given by e.g. micro-electromechanical systems (MEMS) or crystal oscillators, in sensor and FCC do not have the exact same frequency. Also in general, the advertised 20ms cycles of the devices are not exact with respect to physical time (as defined by the SI second): when the device "thinks" (by counting internal oscillator cycles) that 20ms have passed, in reality $(1 + \alpha) \cdot 20\text{ms}$ have passed, where α is a small number with $|\alpha| \lessapprox 10^{-5}$. [Mi17] If the components' value of α are not by chance exactly the same, drifting occurs.

In the plot of sensor-sending and FCC-cycle start events below it can be seen, that typically data from the sensor is not immediately processed by the FCC, as the respective events are not synchronized in any way. Instead, the data is put into some receive buffer at the FCC but is used only some time later, thereby introducing an avoidable dead time into the control loop.

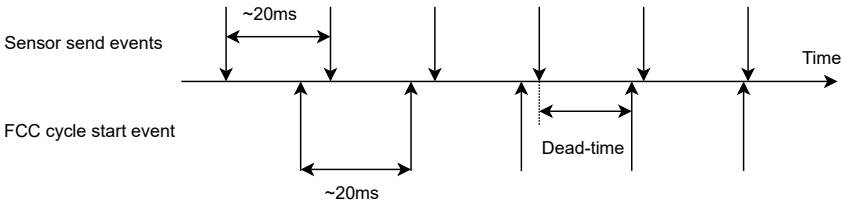


Fig. 2: The drift (resulting in the fact that the dead time is in this case increasing over time) is exaggerated. The duration of the sending of the sensor data packages is idealized to a single event, in reality, this also takes some time.

This leads to reduced stability margins, degrades the control loop performance and additionally introduces an element of uncertainty, as the initial dead time as well as its drift over time is unknown to the controller and has to be dealt with in a "worst-case-way" i.e. always assuming the worst case dead time.

Ideally, the FCC would start its cycle directly after new data from the sensor is received and ready to be used. This would result in a purely event-triggered real-time system [H.11], where the paradigm is that successful reception of a new data package of the sensor directly triggers an interrupt that then starts the execution cycle. However, this is in some applications not a viable solution:

- It could be that the sensor has a considerable amount of jitter or an other type of irregular sending behaviour - however still maintaining the 20ms update rate on average. This irregularity would then directly propagate further into the timing of the control loop. ([M.21], chapter 4).
- From experience, most existing flight control software infrastucture is based on the time-triggered principle and employ some form of RTOS that relies on a 1ms timer. To change the paradigm to event-triggered would be a major undertaking on the existing code base. Also related to certification, the lesser direct external influence there is on the computing hardware, the more deterministic behaviour can be expected from the system.
- In some FCC architectures, there is a dedicated I/O processor or FPGA as well as a number-cruncher, that executes the algorithm. Those two units are not connected via an interrupt line and so the main processor can not be immediately notified about a reception event. However, timing data can be forwarded from the I/O processor and the number-cruncher can from that calculate the dead time, which is the only relevant data for the proposed algorithm.

Other proposals to deal with this problem are to inform the controller about the current value of the dead time [P.01] and afterwards adjust the controller parameters on a per-cycle-basis.

Some sensors also offer the possibility to request a new data package via a dedicated Sync-line input. [Ve17] However, for typical COTS components, this is not the case and typical sensors asynchronously broadcast their data.

Another solution to this problem would be the use of a global time triggered architecture, where all devices agree with a high precision on a global time base and perform their actions completely synchronous to each other. This involves exchanging dedicated messages whose only purpose is the synchronization of time. [H.11] Also, all devices have to support this kind of architecture by offering respective hardware and software interfaces. This is generally not the case for COTS components.

This paper describes the current development state of the pragmatic method used to solve this problem in flight control systems developed at the Institute of Flight System Dynamics at the Technical University of Munich.

In section 2, the main idea for the proposed algorithm is described in a simplified setting, section 3 presents the simulation framework and results. Section 4 talks about the experimental implementation and section 5 gives an outlook on further work to do.

2 Relative synchronization algorithm

Pictorially speaking, the algorithm will execute on the FCC and make out of fig. 2 the following, while still keeping up the time-triggered software structure:

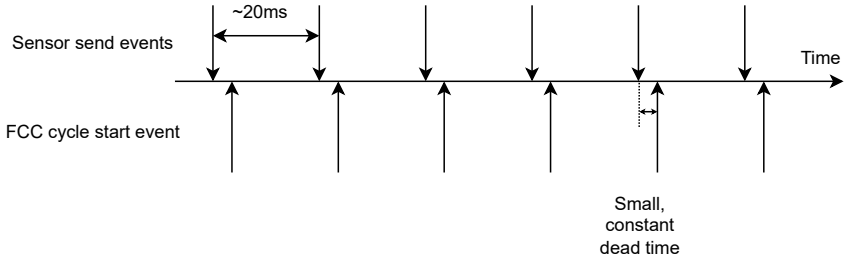


Fig. 3: Situation with synchronization: the dead time is small and constant over time

Properties of the algorithm will be

- "Minimal invasive": The existing code base does not have to be extensively manipulated, only two additional functions have to be added: One interrupt and a small function that executes directly before the controller and performs the synchronization task. As no extra dependencies are introduced, this almost feels like "free lunch".
- Easily adaptable and applicable to a multitude of hardware environments (processors, interface chips, COTS internal architecture, data buses).

- Fail passive: The algorithm can seamlessly deactivate its influence on timing and by that revert to the usual way of scheduling when unexpected behaviour arises. Now, any unexpected behaviour in the reception of relevant sensor data is an indication of a critical problem in the communication between sensor and FCC and is by itself a reason to deactivate the high-performance flight control law. So this information should be passed directly to the functional input monitoring on the FCC and appropriate action has to be triggered, for example switching to a backup flight control lane that uses a different set of hardware units as well as a simpler emergency flight control law. This is the typical structure of system architectures developed at FSD. Deactivation of the algorithm guarantees that the FCC continues its execution cycles seamlessly, providing its remaining functions and output (like forwarding that the sensor has failed), although not being control of the aircraft anymore.

The main principle is the following: A free running timer is activated on the FCC. Its prescaler is set such that the granularity is in the range of some μs. As explained above, the FCC-internal notion of time does not coincide with the SI-definition of time. Instead, in a good approximation, the physical time Δt_{ph} that has passed when the FCC-measured oscillator time Δt_{FCC} has passed is different by a factor of $1 + \alpha_{\text{FCC}}$. The same holds for the sensor.

$$\Delta t_{\text{ph}} = \Delta t_{\text{FCC}} \cdot (1 + \alpha_{\text{FCC}}) \qquad \Delta t_{\text{ph}} = \Delta t_{\text{sensor}} \cdot (1 + \alpha_{\text{sensor}}) \tag{1}$$

Further, an interrupt that will be entered when a relevant message from the sensor has been received, will be installed. The interrupt handler is very short: only the value in the timer register at the moment of the interrupt will be noted down, as well as the fact that a message was received. Then, at the beginning of the FCC cycle, the time of the last reception of a message and the current time is compared (this is the dead time in FCC-internal time units) and then a control problem is solved: control this value to a small constant.

The relevant control variable is related to the way the FCC together with the time-triggered RTOS keeps track of its internal 20ms cycle. On the processor, there typically is a dedicated periodic interrupt timer that every 1ms generates an interrupt and in it increases the system time and possibly schedules a task that after the time tick has now highest priority. This "train of interrupts" is derived from the internal high speed processor clock which is after prescaling fed into a periodic interrupt timer. At every prescaled clock event, the timer value is increased and when it reaches a certain value K_0, it is reset to zero and an interrupt is triggered. The constant K_0 is chosen such that together with the fixed prescaling, an interrupt is triggered every 1ms.

By changing K_0 by small amounts in the range below 1% to some other value $K = K_0 + \Delta K$ the duration of a time tick on the FCC can be adjusted, while being completely transparent to the rest of the software. Note that the internal oscillator frequency will (and can) not be

changed, only the construction of the internal timing variable from this oscillator will be adjusted.

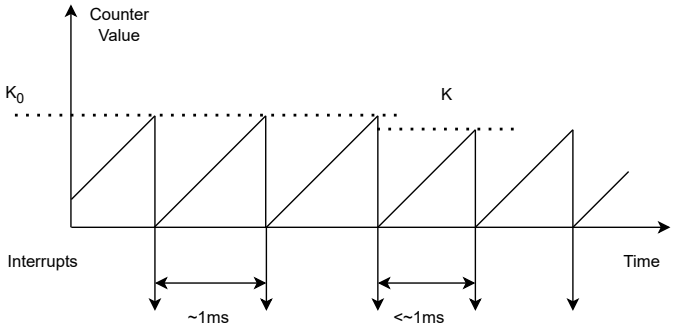


Fig. 4: Internals of the periodic interrupt timer: the discrete nature of the counter value is not displayed in this figure, but typical values for K are in the range of 1e6. The adjustment of K is also exaggerated

In formulas, the physical time that passes between two FCC cycles dependent on K is

$$\Delta t_{\text{ph}} = (1 + \alpha_{\text{FCC}}) \cdot 20ms \frac{K}{K_0} \tag{2}$$

For explanation simplicity, we now assume, that there is no jitter in the system, that the parameters α_{FCC} and α_{sensor} are constant over time and the situation is always such, that the sensor receive event always happens “in the middle” between two FCC cycles.

To get an explicit equation that describes the behaviour of the dead time under the influence of the control variable K, we have a zoomed in look at the left part of fig. 2:

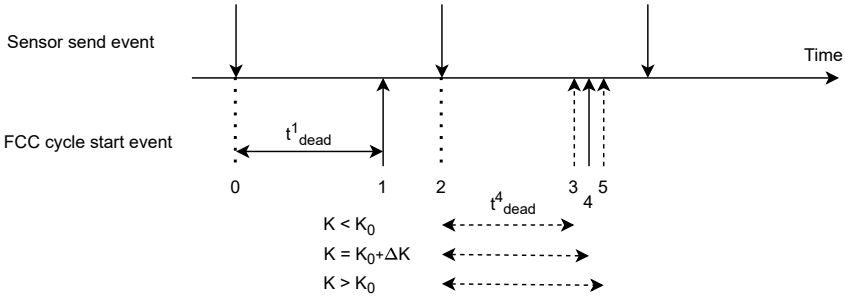


Fig. 5: By changing the constant K, the next FCC cycle start can be adjusted and by that the next dead time can be changed.

Let's assume that the FCC is currently at time 1 and the cycle has just started. The last sensor send event at time 0 was timestamped at the FCC by the interrupt and the current time 1 can be read in the internal timer register. So the FCC measures the most recent dead time at time 1 to be

$$\mathrm{t}^1_{\mathrm{dead}} = \frac{t_1 - t_0}{1 + \alpha_{\mathrm{FCC}}} \tag{3}$$

If there is a correction applied to K by the FCC, the next cycle will start at time t_4 which is by (2) related to t_1 via

$$t_4 = t_1 + (1 + \alpha_{\mathrm{FCC}}) \cdot 20\mathrm{ms}\frac{K}{K_0} \tag{4}$$

By setting $K = K_0 + \Delta K$, this can be rewritten as

$$t_4 = t_1 + (1 + \alpha_{\mathrm{FCC}}) \cdot 20\mathrm{ms}\frac{K}{K_0} = t_1 + (1 + \alpha_{\mathrm{FCC}}) \cdot 20\mathrm{ms} + (1 + \alpha_{\mathrm{FCC}}) \cdot 20\mathrm{ms}\frac{\Delta K}{K_0} \tag{5}$$

By (1), the next sensor send event will happen at t_2

$$t_2 = t_0 + (1 + \alpha_{\mathrm{sensor}}) \cdot 20\mathrm{ms} \tag{6}$$

So the next dead time measured by the FCC at time 4 will be

$$\mathrm{t}^4_{\mathrm{dead}} = \frac{t_4 - t_2}{1 + \alpha_{\mathrm{FCC}}} \tag{7}$$

By plugging in the expressions for t_2 and t_4 obtained above, this evaluates to

$$\mathrm{t}^4_{\mathrm{dead}} = \frac{1}{1 + \alpha_{\mathrm{FCC}}} \left(t_1 - t_0 + (\alpha_{\mathrm{FCC}} - \alpha_{\mathrm{sensor}}) \cdot 20\mathrm{ms} + (1 + \alpha_{\mathrm{FCC}}) \cdot 20\mathrm{ms}\frac{\Delta K}{K_0} \right) \tag{8}$$

and finally by using the definition of t^1_{dead}:

$$\mathrm{t}^4_{\mathrm{dead}} = \mathrm{t}^1_{\mathrm{dead}} + \frac{\alpha_{\mathrm{FCC}} - \alpha_{\mathrm{sensor}}}{1 + \alpha_{\mathrm{FCC}}} \cdot 20\mathrm{ms} + \frac{\Delta K}{K_0} 20\mathrm{ms} \tag{9}$$

The contribution $\frac{\alpha_{\text{FCC}}-\alpha_{\text{sensor}}}{1+\alpha_{\text{FCC}}} \cdot 20\text{ms}$ comes from the relative drift of the two devices, while the contribution $20\text{ms}\frac{\Delta K}{K_0}$ is something the FCC can influence. So the control problem for t_{dead} has the following simple form:

$$x_{n+1} = x_n + D + u_n, \quad \text{where } D = \frac{\alpha_{\text{FCC}} - \alpha_{\text{sensor}}}{1 + \alpha_{\text{FCC}}} 20\text{ms} \quad \text{and} \quad u_n = \frac{\Delta K_n}{K_0} 20\text{ms} \tag{10}$$

Now, different suitable control algorithms can be thought of, including for example PID-type controllers (with appropriate bounding of ΔK_n to stay within $\pm 1\%$ of K_0) or constant ΔK_n controllers to reach the desired set point of the dead time. To minimize jitter in the synchronized state, u_n can then also be chosen to be $-D$, then the FCC and sensor have in fact synchronized their execution cycles.

To calculate the control, the parameter D has to be known with reasonable precision. To achieve this, the algorithm performs an on-line real time estimation from reception timing data gathered from the sensor. In easy words, because a relative drift D results in the 20ms cycles of the 2 devices not being the same, measuring the resulting duration of a "20ms sensor cycle" in local FCC time should make it possible to make a statement about the value of the drift. And indeed that is true:

By (6) the physical time difference between two sensor send events is $\Delta t^s_{\text{ph}} = (1+\alpha_{\text{sensor}})20\text{ms}$. Because each of those triggers an interrupt on the FCC, where the respective timestamps of the reception are taken, the FCC can in every cycle obtain a measurement of Δt^s_{ph} by subtracting the last reception time from the most recent one. In terms of fig. (5), at time t_4, the FCC subtracts its measured values of t_2 and t_0. Again, by (1), the difference time measured on the FCC will be

$$\Delta t^s_{\text{FCC}} = \frac{t_2 - t_0}{1 + \alpha_{\text{FCC}}} = \frac{1 + \alpha_{\text{sensor}}}{1 + \alpha_{\text{FCC}}} 20\text{ms} \tag{11}$$

By subtracting this value from the expected 20ms, the FCC can in every cycle compute a candidate for D:

$$20\text{ms} - \Delta t^s_{\text{FCC}} = 20\text{ms} - \frac{1 + \alpha_{\text{sensor}}}{1 + \alpha_{\text{FCC}}} 20\text{ms} = \frac{\alpha_{\text{FCC}} - \alpha_{\text{sensor}}}{1 + \alpha_{\text{FCC}}} 20\text{ms} = D \tag{12}$$

This computation will be done in every FCC cycle to get:

$$D^m_i = 20ms - \left(\Delta t^s_{\text{FCC}}\right)^i \tag{13}$$

As some amount of jitter is involved in a real system however, the obtained values D^m_i will be different in every cycle i. However, taking the mean of a large number of measurements

will improve the quality of the estimate. A suitable algorithm with low demands on memory is the exponential moving average algorithm, where the estimate D^e for D is updated according to

$$\begin{aligned} D_0^e &= D_0^m \\ D_{i+1}^e &= \lambda D_i^e + (1-\lambda) D_i^m \end{aligned} \tag{14}$$

with some "forgetting factor" $0 < \lambda < 1$. In the end, the respective α parameters don't have to be known, but luckily their relevant combination - the parameter D - can directly be measured.

3 Simulation

A simulation framework that helps to test, assess and validate a candidate algorithm in different settings (also accounting for facts that were ignored above, like jitter) was developed in MATLAB. For this, the timing behaviour of sensor and FCC can be specified by their parameters α, the magnitude of the jitter involved, and an initial offset between the two devices. The simulation framework emulates the hardware behaviour of the components with their clocks and calculates e.g. the counter values of the reception events at the FCC as well as the correction to the value K using a certain type of controller for (10) that is also implemented in MATLAB. Plots of the evolution of different properties over time can then be created. As an exemplary output, here we see the evolution of the dead time when there is significant drift and jitter in the sensor:

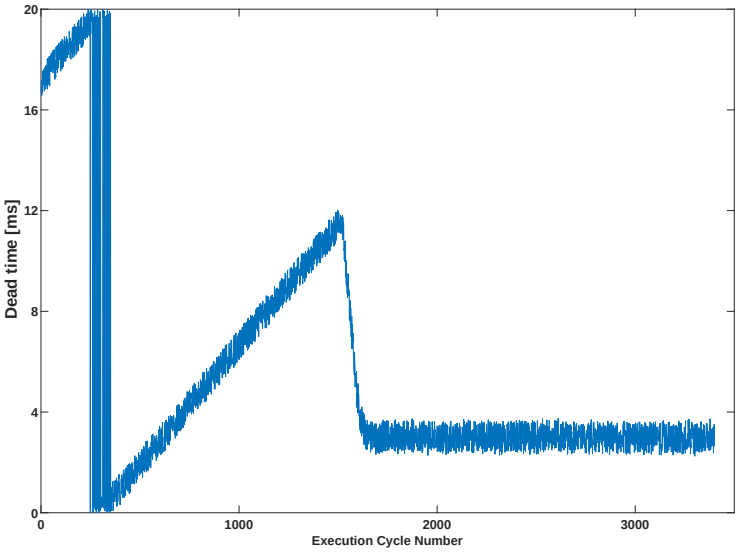


Fig. 6: Simulated evolution of the dead time. Up until cycle number 1500, only the drift estimation module was active, while afterwards, the synchronization controller was activated. Here a simple P-controller was used. The dead time is afterwards stable at the small value of 3.5ms

Furthermore, graphic output like in figures (2) or (3) can also be created.

The simulation environment is planned to have an interface to Simulink System Composer, such that relevant device and interconnection information can be extracted from a system architecture at the design stage. Further, the environment allows for connection of multiple sensors to one FCC, as well as e.g. the mutual connection of three redundant FCCs that should synchronize to each other. Also, a cascaded scheme can be created, which is for example needed, when the FCC forwards its data to an actuator interface unit, which should also be synchronized to the FCC. As another future step, device fault conditions (like a reboot that changes the cycle offsets and clears the drift filter states, or device malfunctions that result in no messages sent at all) will be included.

4 Implementation

In the flight control system for the EPUCOR helicopter [B.22], there is a COTS-type flight control computer that is directly connected to a COTS-type INS sensor, as described in fig. 1. The relevant part of the FCC consists of a STM32F405ZG microprocessor [ST19] and an

external A429-Interface chip, a HOLT3210 [HO17]. Instead of using the real INS sensor from the FCS - an AHR 150A from Archangel [Ar17] - a mockup device was used. By that, parameters like jitter and drift of the sending unit can be easily set and manipulated to test the performance of the algorithm.

The message structure of the AHR 150A is such, that a number A429-Labels are sent directly after each other. Then the bus is idle until the 20ms cycle starts over. The transmission cycle always begins with labels 331, 332, 333, 327, 326, and 330. These happen to be the angular rates and specific forces - the immediately most relevant for flight control.

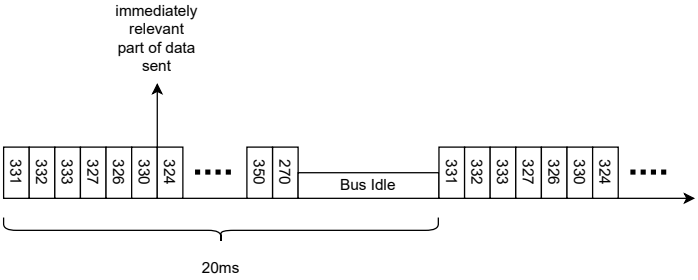


Fig. 7: Packet structure of the AHR 150A

For time recording purposes on the FCC, the basic timer TIM6 is used. The FCC's timing source itself is a 8 MHz Quartz Crystal, whose frequency is upscaled to 168MHz with a PLL. Via the configurable prescaler for TIM6, the uint16-counter increases its value by 1 every μs and reloads to zero at the next tick when it has reached its maximal value $2^{16} - 1$.

The HOLT 3210 offers the possibility to signal an interrupt to the Microcontroller on the reception of a certain label that is configurable in an interrupt table. By setting the corresponding entry for label 330 to 1, the time where a full relevant INS message is available will be automatically noted down.

The timer that creates the 1ms interrupt for the RTOS directly comes from the processor-integrated SysTickTimer that counts with a frequency of 168MHz, where the nominal value of K is 168000.

The integration into the already implemented periodic control task is achieved by calling the synchronization function directly before the input parsing:

```
while(1)
{
    wait_for_20ms_cycle_start();
    estimate_D();
    calculate_and_set_new_K();
    parse_inputs();
    execute_controller();
    send_outputs();
    put_task_to_inactive();
}
```

The relevant part `estimate_D()` essentially consists of equations (13) and (14), whereas `calculate_and_set_new_K()` implements some controller for the problem (10). This procedure is very general and demonstrates the universality and easy realization of the proposed algorithm.

For demonstration, as in the simulation in fig. (6), a P-controller and a set point of 3.5ms was chosen. In the plot of the dead time over execution cycle, a comparison between the simulation and the real implementation shows a excellent agreement and tells that the simulation is of very high fidelity and suitable to further continue the development process based on the digital twin.

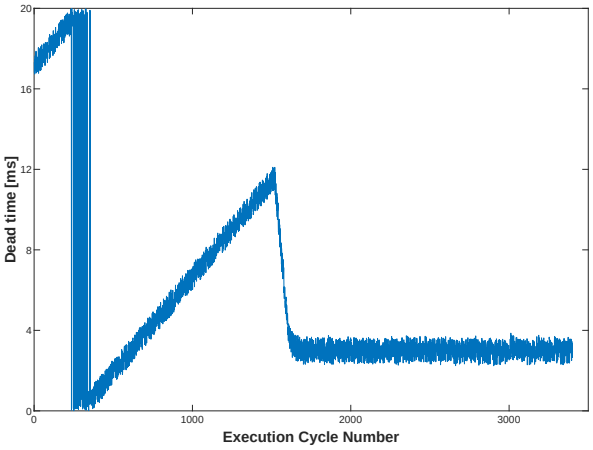


Fig. 8: Experimental data taken from the implementation. From execution cycle 1500 on, the controller was activated. Note the similarity to fig. (6)

5 Conclusion and Outlook

In this short paper, the need for an easy to realize synchronization for COTS components was illustrated. In the end, this task can be accomplished by solving the control problem 10. The comparison between a real implementation and simulation in a dedicated environment shows very good agreement. The algorithm is adaptable to many different hardware settings and up to now successful implementations have been demonstrated using three different further FCCs that are currently used at the institute of flight system dynamics at TUM, with different interfaces like CAN and RS-422 and also using different interface chips or integrated processor peripherals. Also, instead of using the "mockup-device" as sender, an AHR 150A as well as a VectorNav VN310 were used - the synchronization results were as expected.

Compared to existing solutions for the general problem of "time synchronization in distributed systems", the main differences are

- **Hardware**: No dedicated hardware that goes beyond "standard equipment" is needed. Compare that to time triggered system architectures, where special interface chips are needed, like e.g. the AS8202B for TTP or the NXP MFR4310a for FlexRay. Further, no elements whose sole purpose is to establish synchronization have to be introduced into the system architecture, as it is for example the case for the sync-lines in the space shuttle flight control system [Jo89]. This allows for subsequent updates of existing flight control systems.
- **Algorithm**: For the majority of established existing clock synchronization algorithms (examples being [J.88], [Ma95], or the network time protocol NTP) their purpose is to construct from the local times of the components a global time that is within some bound the same for all involved components. This is done by adding to the respective local times some correction values that represent the offset to the global time . For that, dedicated messages have to be exchanged and a so-called "convergence function" is applied to locally compute the correction. In the approach described in this paper, the main goal is not to obtain a common notion of absolute time on different components, but rather to synchronize their execution cycles to minimize dead times. This is mainly done by adjusting the clock rate and not by adding correction values. Also, no dedicated messages are used, but the relative timing information is obtained from the ordinary communication that takes place.

However, In the discussion above, there were some points that were for the sake of simplicity ignored. For example, the effect of jitter and dead times near zero lead to the fact, that in some FCC cycles no messages at all are received, while in the adjacent cycles, 2 messages are received or a very big naively calculated value for the dead time is obtained. This corresponds to the thick vertical lines in fig. 8.

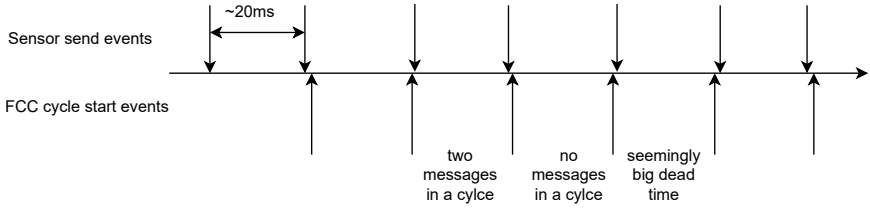


Fig. 9: Behaviour under the influence of jitter in an “almost” synchronized state

Future investigations that deal with this fact will be conducted and the overall goal is to prove robustness properties of the unconditional stability of the algorithm either by mathematically analyzing the control problem or using formal methods like NuSMV, where some unconditionally true results could already be obtained.

Acknowledgements

This work was funded by the Federal Government of Germany as part of the LuFo program (funding ID: 20Y1705C).

Supported by:

Federal Ministry
for Economic Affairs
and Climate Action

on the basis of a decision
by the German Bundestag

Bibliography

[Ar17] Archangel Systems: AHR 150A Datasheet. 2017.

[B.22] B. Hosseini, F. Sax, J. Rhein, F. Holzapfel, L. Maier, A. Barth, M. Hajek: Global Model Identification for a Coaxial Helicopter. VFS Forum 78, 2022.

[H.11] H. Kopetz: Real-time Systems: Design Principles for Distributed Embedded Applications. Springer, 2011.

[HO17] HOLT: HI-3110 Manual: Rev. L. 2017.

[J.88] J. L. Welch and N. A. Lynch: A New Fault-Tolerant Algorithm for Clock Synchronization. ACM Information and Computation, (77), 1988.

[Jo89] John F. Hanaway, Robert W. Moorehead: Space Shuttle Avionics System. NASA SP-504, 1989.

[M.21] M. Saleab, F. Sax, J. Schumann, F. Holzapfel: Toward Timing and Memory Analysis for UAS Flight Code. AIAA Scitech 2021 Forum, 2021.

[Ma95] Manfred Pfluegl, Douglas Blough: A New and Improved Algorihtm for Fault-Tolerant Clock Synchronization. Journal of Parallel and Distributed Computing, (27), 1995.

[Mi17] Microchip: DSC1001/3/4: 1.8V-3.3V Low-Power Precision CMOS Oscillators. 2017.

[P.01] P. Martí, J.M. Fuertes, G. Fohler, K. Ramamritham: Jitter Compensation for Real-Time Control Systems. Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001), 2001.

[ST19] STMicroelectronics: RM0090 Reference Manual. 2019.

[Ve17] VectorNav Technologies: VN310 User Manual: Rev. 2.47. 2017.

Enhancing System-model Quality: Evaluation of the MontiBelle Approach with the Avionics Case Study on a Data Link Uplink Feed System

Hendrik Kausch [1], Mathias Pfeiffer [1], Deni Raco [1], Bernhard Rumpe [1], Andreas Schweiger [2]

Abstract: Software quality is often related directly to the quality of the models used throughout the development phases. Assuring model quality can thus be an important aspect for assuring the quality of the final product. Measuring model quality is done via different quality indicators. In this article, we investigate the influence of our holistic systems engineering methodology on model quality. An avionics case study was previously conducted using our methodology. The developed SysML v2 model artifacts are evaluated in this paper regarding internal and external model quality, as well as model notation quality. In total, the positive impact on 26 model quality indicators from our previous work is argued. These indicators are divided into intra-model (single artifact) quality indicators and inter-model (across model artifact) quality indicators. The inter-model quality indicators are further classified into indicators for models at the same granularity level (horizontal) and across several granularity levels (vertical). Multiple quality indicators are positively affected by the modeling language's capabilities and the underlying mathematical semantics. Other indicators depend on methodological guidelines that steer the engineering process. The evaluation of model-quality properties leads towards maturing a holistic systems engineering methodology that facilitates high model quality and thus indicates high product quality.

Keywords: Model Quality; Design Methodology; Theorem Prover; Formal Verification

1 Introduction

1.1 Model Development in Computer Science

It can be observed that the size of software increases continuously [Le97]. Additionally, the degree of its complexity grows. These developments make the full understanding of software systems more difficult. In order to maintain control over such complex software systems, adequate mechanisms are necessary. These include, e.g., approaches according to the principle divide et impera (Latin) or a suitable model usage. Model building represents the essential foundation of the tasks of computer science [BS04, Br19, Br23], since the representable, processable, storable and transferable representation (syntax) of information must always be subjected to interpretation in order to derive the intended semantics. To achieve this, syntactic representations abstractly represent the object of consideration, in order to represent the characteristics relevant for the application purpose. Accordingly, models usually exhibit the following properties [St73]:

[1] RWTH Aachen University, Chair for Software Engineering, Aachen, Germany
[2] Airbus Defence and Space GmbH, Manching, Germany

doi:10.18420/sw2024-ws_09

- **Mapping**: Models are always related to an original, which they depict.
- **Reduction**: Models generally do not capture all attributes of the original represented by them, but abstract from them as required for the respective purpose or use case.
- **Pragmatism**: Models are not uniquely mapped to their originals per se, rather they fulfill their substitution function for certain subjects under certain restrictions (purpose and boundary conditions of the model construction). Thus models are subject to variability with respect to the mentioned aspects.

Furthermore, models can be differentiated according to their purpose:

- A **descriptive** model describes an original for understanding or analysis (e.g., a city map).
- A **prescriptive** model describes the way of making of an original (e.g., a construction plan).

In addition, significant value is placed on the **visualization** of models. This is supported by the wide acceptance of semi-formal modeling languages in both the academic and industrial domains. Examples are UML for software or SysML for systems development. The introduction of such models alone will not be able to make the desired contribution to mastering complexity, unless the models have the quality properties relevant to the development project at hand. A model is the most important artifact of model-based development, and its quality thus contributes decisively to the success of the project [FHR08]. For this reason, quality assurance guidelines such as those in the IEC 61508 standard [IE10] rightly demand adequate quality of models in the development process. In particular, quality requirements for models also include requirements for the modeling notation in use. In Tab. 1, quality properties for models according to [FHR08] are summarized and explained. This taxonomy forms the basis for the evaluation (see Sect. 3) of the MontiBelle approach for model-based development and formal verification presented in Sect. 2.

Tab. 1: Overview of quality properties for models [FHR08, p. 416-420] with extensions of vertical quality properties redundancy and controlled redundancy, as well as additional quality properties verifiability and transformability

Intra-model quality properties	
Representation	Representation measures how easily information can cognitively be comprehended. A better representation yields better understanding. Aesthetically pleasing and well-structured models are easier to understand.
Precision	Precision measures whether all relevant properties of the modeled system have been captured. Precision addresses the reduction of information as a result of modeling.
Universality	Universality measures whether only relevant details are modeled. A low universality could result from unnecessarily fixing platform-specific details in early design phases and could, e.g., induce a complex solution in hardware later on.
Simplicity	Simplicity measures whether relevant details are not modeled any more complex than necessary. Simplicity can be increased without losing any information content, e.g., by reformulating, restructuring, and using abstraction.
Semantical adequacy	Semantical adequacy expresses the suitability of a model to purposefully represent desired information. For example, entity-relationship-models are well suited to represent entities and their relations. They are less suited to model the behavior of software components.

Continued on next page

Tab. 1 – Continued from previous page

Consistency	Internal Consistency measures the absence of errors within a single model. A typical consistency check verifies that variables are defined before their use.
Conceptual integrity or uniformity	Conceptual integrity or uniformity aims at providing similar solutions to similar problems. This quality indicator measures the similarity of modeled solutions within single models. Conceptual integrity can be achieved by applying repeatable rules, patterns, and principles.
Conformance	Conformance measures the level of application of standards and norms to models. Conceptual integrity can thus be a consequence of conformance, assuming the standards and norms have relatively tight boundaries. Typically, project specific guidelines are required to achieve conceptual integrity from conformance.
Language-specific, semantical quality properties	These quality properties include any additional quality indicators that the specific modeling language provides. Examples include completeness of state charts or liveness in Petri nets [Re85].
Horizontal inter-model quality properties	
Consistency, conceptual integrity, language-specific, semantical quality properties	Intra-model quality properties in the inter-model perspective. For example, matching interfaces, i.e., the description of visibility along with management and transfer of interface elements, are necessary.
Downward completeness	Downward complete models must contain all necessary information for developing model artifacts of the next development step in the model-chain.
Cohesion	For cohesive models, interrelated parts and facts must be modeled in closely related model artifacts.
Modularity	Modular models must represent single aspects to facilitate re-usage of model artifacts.
Freedom from redundancies	Even though complete redundancy-free modeling is not possible or desired (see controlled redundancy), redundancy between models must be minimized.
Controlled redundancy	Allowing purposeful redundancies in models can be leveraged , e.g., for different views of system parts or facts [Gr08].
Vertical inter-model quality properties	
Correctness	Correctness measures whether a model implements requirements from previous artifacts in the model chain rightly. This could include that requirements of previous stages are developed towards a correct implementation.
Downward correctness	A granularity level of models is downward correct, if all requirements of this level's artifacts are refined in subsequent granularity levels.
Upward completeness	Upwards completeness measures how complete the requirements of a previous development phase were adhered to. Upwards completeness is highly related to correctness, as only correct derivations may result in completeness. However, correctness does not induce completeness. Consider requirements of a previous stage to be implemented in multiple new models. Correctness can be measured for each new model individually. In contrasct, completeness requires consideration of the whole model set.

Continued on next page

Tab. 1 – Continued from previous page

Traceability	Traceability measures the ability to track which information from one model was used to create (parts of) a new model in a subsequent development phase.
Modifiability	Modifiability measures the ability to maintain (for correcting and for evolving), extend, and re-use models or model elements.
Freedom from redundancies	To achieve freedom from redundancies over subsequent granularity level's artifacts, subsequently developed models should not model previously modeled information again.
Controlled redundancy	Introducing controlled redundancies over multiple granularity levels, i.e., modeling the same properties or behaviors in different ways, allows checking semantical consistency of requirements between successively developed model artifacts.
Quality properties for modeling notation	
Degree of formalization	The degree of formalization measures whether and to which extent the models adhere to a formal language. An increased extent of formalization increases the ability to analyze, simulate, or generate artifacts from models. Later development phases generally require higher extent of formalization.
Adequacy for the application domain	Measures whether and how well a modeling notation can be used to model typical domain concepts. Models should generally be clear and compact, while still maintaining the ability to model even complex domain specific concepts. Non adequate notations typically lead to unnecessarily complex models or even non modelable concepts.
Other model quality properties	
Verifiability	Verifiability determines if and how well modeled properties can be verified regarding their correctness. Verifiability requires a certain extent of formalization, as models and their properties need clear semantics for verification to take place. A special case of verification is testing, which requires an executable model.
Transformability	Transformability measures whether and how easily models can be processed and transformed, typically but not necessarily by machines. Transformations can be used to derive simulations or other (formal) analysis.

1.2 Contribution

As described in Sect. 1.1 model-based development provides an indispensable means for mastering increasing complexity of software in general and in particular in the avionics domain. This in turn has a positive impact on the product quality. Since the product quality is determined by the models' quality, we have to provide guidance on how to reach said quality.

Bansiya and Davis [BD02] developed an approach for quantitative assessment of the design properties in object-oriented designs. From the corresponding metrics they calculate the high-level quality attributes. However, the approach is valid for object-oriented systems, while model-based development is the more general approach, which we are targeting in this contribution.

In addition, since some system properties cannot be verified exhaustively by testing only and others are amenable to verification only by the deployment of formal methods, the model-based approach needs to cover the integration of formal methods, as well. To the best of our knowledge, no such method has been presented yet. The contribution of this paper is answering the following research questions:

- **RQ (1)**: Which quality attributes of models are relevant for determining the product's quality?

- **RQ (2)**: To which extent does the selected model-based approach with formal underpinning for precise semantics support the development of models of high quality?

1.3 Structure

RQ (1) has been answered with the proposal of the attributes described in Tab. 1, where these are applicable to any model-based development approach. Sect. 2 covers the MontiBelle approach. This provides the required semantical underpinning via FOCUS, offers a tool chain, includes a development methodology for mitigating the risks of formal verification, and increases the success rates of such a verification. Using an avionics case study in Sect. 3 we evaluate to which extent MontiBelle addresses the identified model quality properties. This answers **RQ (2)**. We conclude in Sect. 4 and describe needs for future development in Sect. 5.

2 MontiBelle Approach

The approach under evaluation in this paper is MontiBelle [Ka20b]. It provides a framework for model-based formal verification. The framework consists of (1) FOCUS [BR07] as a semantic foundation for the interpretation of system models, (2) a methodology for system development, and (3) a tool chain.

2.1 FOCUS

In the formal specification language FOCUS, distributed and interactive systems consist of components that exchange messages over unidirectional channels. The semantics of a component is a (set of) stream processing functions, each of which represents one potential behavior. The refinement of a component's behavior is represented by set inclusion ($\subseteq$) between the original and current components' semantics. Concurrency is represented by an appropriate composition operator that connects channels. The most important reason for using FOCUS is the property that refinement is fully compositional [BR07, Ka20b]. This means that if a system has been decomposed and the parts were refined separately, then the reassembled refined parts are by design a correct refinement of the system before its refinement.

2.2 Methodology

The MontiBelle methodology is a development approach derived from SPES [Bö21] and SpesML [Ge23]. It fulfills the requirements of EUROCAE ED-216, which describes the use of formal verification in the development of software systems for the avionics domain. The MontiBelle methodology aims at providing users with the necessary guidelines to intuitively and successfully apply formal verification to system engineering tasks. MontiBelle uses model-driven concepts to enable abstraction where possible. At the same time, formal specification techniques allow fine-grained control over system specifications where necessary. The MontiBelle approach covers the early stages of development, as well. As such, it provides means to ensure conformance, consistency, verifiability, and traceability between system requirements (SRs), high-level requirements (HLRs), and software architecture design and low-level requirements (LLRs).

First, typically informal SRs are formalized in HLRs as declarative specifications over communication histories. We use input-output relations in assumption-guarantee-style[3]. Designing a system is, in accordance with EUROCAE ED-216, divided into two activities: deriving an architecture and developing LLRs. The MontiBelle approach proposes to decompose HLRs into communication architectures of more detailed and specialized HLRs, until a fine-grained enough architecture is reached. When refining declarative specifications to either other declarative specifications or to architectures [PR97], then meeting the resulting proof obligation typically entails showing the implication between (potentially multiple and coupled) logic predicates. The compositionality of refinement in FOCUS and MontiBelle's toolchain automates the proof-finding. The LLRs are formalized using prescriptive models. MontiBelle suggests event-driven automata [PR94, Ka23] to model LLRs as they are implementation-oriented and specifically fitting for software-intensive systems. One of their most important properties is the fact that they are implementable by construction. Lastly, architecture and LLRs are combined into a system design. This is achieved by refining the final HLR architecture to an equally structured architecture composed from LLR. The proof for correctness of the refinements follows directly from the compositionality of FOCUS and is fully automated.

2.3 Avionics Case Study

The development of an avionics data link was previously conducted using the MontiBelle approach [Ka22, Ka23]. This Data Link Uplink Feed (DLUF) system is representative for software systems in the avionics domain and will be used to evaluate the claim of increased model quality. The DLUF system should enable components using a wireless connection (e.g. between an Unmanned Aerial Vehicle (UAV) and its ground station) to transfer prioritized data packets. The objective was to develop a system that adheres to a set of 18 system requirements. It required to formally verify (instead of just demonstrating the correct functionality with non exhaustive tests), that these properties hold for the overall system (instead of just subsystems) in every scenario (instead of just best case scenarios). One of the requirements, the non-starvation (or liveness) property, required the use of formal methods to ensure a correct DLUF system. This property could not be tested for, as it required checking an unknown and potentially infinitely long time frame. It had to hold for the overall system and could not be sufficiently verified by only checking properties of the system's parts, but required the integration of all artifacts into a single coherent claim.

According to the MontiBelle methodology, the non-starvation requirement was encoded in a descriptive model using the textual notation of SysML v2. The specification is contained in a SysML part definition. The interface is modeled using port usages of a custom type called Packets. The HLR shown in List. 1 is expressed using a (descriptive) requirement which groups four SysML constraint usages, three of which as assumptions and the last as guarantee, designated with the *require* keyword.

[3] A style of specification, that asserts certain properties (the guarantees), if certain preconditions (the assumptions), hold.

```
part def DLUF_HLR1 {
  port input: ~Packets[4]; port output: Packets[4];

  satisfy requirement 'non-starvation' {
    assumes 'infinitely long timeframe' {
      ∀i∈{1,2,3,4}. input[i].length() = ∞
    }
    assumes 'message for each interval' {
      ∀i ∈ {1,2,3,4}, t:nat: input[i].atTime(t).length() > 0
    }
    assumes 'individual packet size below max. capacity' {
      ∀i ∈ {1,2,3,4}: ∀v ∈ input[i].values(): v < maxCap[i]
    }
    require 'infinitely many outputs on all channels' {
      ∀i ∈ {1,2,3,4}: output[i].messages().length() = ∞
} } }
```

List. 1: Descriptive 'non-starvation' HLR formally modeled in SysML v2's textual notation.

From the HLR, we developed fine grained specifications using decomposition. All subsystem specifications at this stage are described using descriptive models similar to the one in List. 1. This approach is in accordance with EUROCAE ED-216 where from System Requirememts, HLRs are developed. Traceability is achieved using a derivative of the general SysML specialization relation called refinement, denoted by the *refines* keyword. Once we reached a suitable decomposition level, we developed prescriptive (LLR) models for each non decomposed (atomic) subsystem. The descriptive models are traced to their prescriptive counterparts using refinement relations. The LLRs are eight total subsystems, four buffering components and four capacity gates. A prescriptive buffer specification is shown in Fig. 1, using SysML v2's graphical representation for better readability:

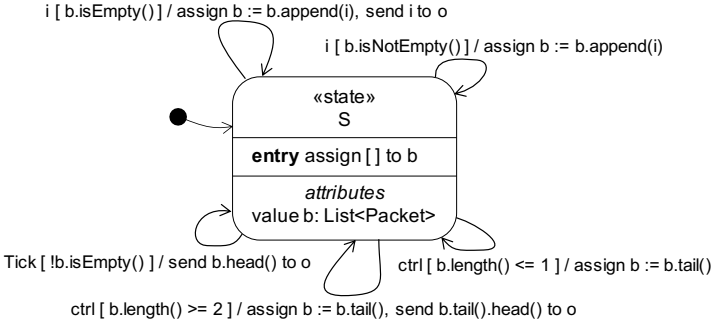


Fig. 1: SysML v2 graphical representation of the prescriptive Buffer model.

The single state contains a list of $Packets$, modeled using SysML v2's attributes. The list stores incoming messages. The transitions handle behavior, i.e., processing incoming messages (top), incoming control directives (bottom right and bottom middle), as well as time passing. Similarly, the capacity gate is shown in Fig. 2.

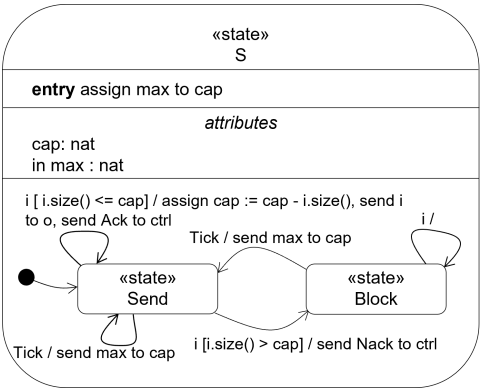


Fig. 2: SysML v2 graphical representation representation of the prescriptive capacity gates.

The capacity gate has two internal states, `Send` and *Block*. The current state shows, whether the current time slice has remaining capacity to send messages. As long as remaining capacity, modeled as the attribute *cap*, is sufficient, messages are forwarded. Once the capacity threshold is reached, messages are blocked until time passes. A control message $Nack$ is sent to inform the buffer component about the blockage. Operation continues, once capacity becomes available. The prescriptive models are assembled into an LLR of the entire system via composition. There are four scheduler subsystems of buffer and capacity gates. A graphical representation of the first subsystem is shown in Fig. 3:

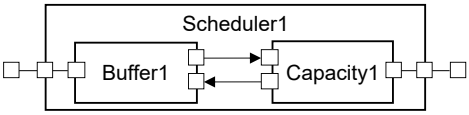


Fig. 3: Graphical representation of a scheduler subsystem from buffer and capacity gate.

The method of decomposing descriptive models and refining them to prescriptive models leads to a tree-structure that describes both the development path, as well as the certification artifacts (proof obligations) required for verification. As part of the case study, all these proof obligations where met in an automated way. Fig. 5 visualizes the tracing relation tree for the DLUF case study.

2.4 Tool Chain

Models of a SysML v2 profile named MontiBelleML describe systems of different granularity or abstraction levels. Between the systems at different abstraction levels, modeled refinement relations indicate proof obligations. A generator translates these models and refinement relations into the syntax of a theorem prover, Isabelle [NPW02]. Isabelle enables machine-assisted and automated proof search. In particular, we use the generated theories together with the general theories (FOCUS encodings) to perform formal verification of the refinement relations. The verification can function autonomously using automation scripts. The result consists of machine-verifiable certificates of correctness or, in cases of errors, counterexamples. The tool chain is depicted in Fig. 4.

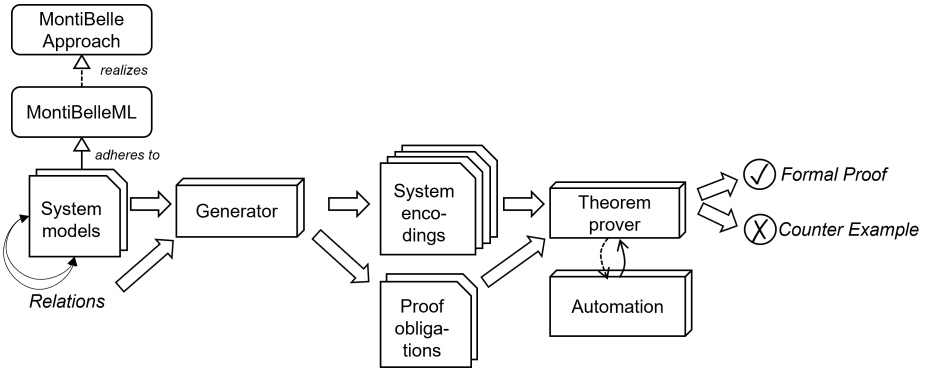


Fig. 4: MontiBelle tool chain overview.

2.5 Related Work

The MontiBelle approach has been applied to case studies from domains such as automotive and aerospace. In [Kr19] time-synchronous modeling was evaluated on an automotive case study. The Isabelle backend was connected to the frontend modeling language MontiArc from the chair of Software Engineering at RWTH Aachen University. MontiArc and a synchronous time model were used again in [Ka20b] for verifying properties of a deterministic pilot flying system from a NASA case study [CM14]. A non deterministic variant of the pilot flying system was refined step by step and proven correct in [Ka20a] using MontiArc and its synchronous time model. Next, SysML was used as frontend to specify the pilot flying system in the synchronous time model [Ka21b], which is more commonly used for hardware modeling. An event-driven modeling approach for distributed software applications was applied in [Ka21a], where the pilot flying system was modeled again using SysML. Using event-driven modeling, another case study from the aerospace domain, the DLUF system, was modeled with SysML and verified by a mapping to Isabelle [Ka22]. Finally, in [Ka23] the DLUF case study was also modeled in MontiArc, where the foundations of event-driven processing using FOCUS were also elaborated.

Formalisms such as Communicating Sequential Processes (CSP) ([Ho85], as used in e.g. [ML09]), Calculus of Communicating Systems (CCS) [Mi82], π-calculus [Pa01], Ptolemy [Le16], Temporal Logic of Actions (TLA) [AL94], Petri nets [Re85] or FOCUS [BR07] are usually used as mathematical underpinning for reasoning. They are typically chosen, because they support non determinism, underspecification, and a notion of behavioral refinement. They further enable the treatment of time-sensitive specifications and hierarchical decomposition. In particular, decomposition is badly needed in general, otherwise the verification of a complex atomic component can quickly become unfeasible because of the computational effort caused by state explosion. This in turn requires compositional[4] verification, which is provided by FOCUS.

Modeling languages can be used to abstract from the complexity of the mathematical formalisms. A number of modeling languages such as Esterel [Be00] or Lustre [Ca87] (and its dialect SCADE) have been created for the development of reactive systems. They are, however, rather suited for the description of hardware systems due to their time-synchronous paradigm. Further methodologies and accompanying tools for specifying distributed systems have been developed, such as the Palladio

[4] Compositionality is introduced by Carnab as *Frege's principle* [Ca47, p. 120-121].

Component Model [BKR09], MechatronicUML [Dz16], AutoFocus [VZ14] or Ptolemy [Le16]. Neither of them supports event-driven specifications or the latest version of SysML, an industry proven and approved modeling language. We chose SysML [Sy23], because it is prominently used in the aerospace and automotive industry for systems engineering.

Lastly, integrating formal verification, particularly deductive methods and modeling languages is not new. The modeling language RSML^{-e} has been combined with the theorem prover PVS [RJH03] via a code generator similarly to our approach. However, the modeling paradigm is synchronous and not event-driven. The modeling language is not an industry standard such as SysML. There exists no automations, as the proving process is manual.

3 Evaluation

The following sections evaluate the MontiBelle approach in regards to our previously introduced quality indicators. We will determine what quality properties are covered by the MontiBelle approach and how so.

3.1 Inner Quality

This section evaluates model quality indicators that are defined for each model individually. Those indicators are called inner quality indicators, because they are defined for single model artifacts. In Sect. 3.2 and Sect. 3.3 quality indicators spanning multiple model artifacts are assessed.

Presentation The MontiBelle methodology suggests the usage of decomposition of specifications. MontiBelle also suggest the use of abstraction and underspecification. Both lead to simpler and more managable model artifacts as demonstrated by the decomposed *Scheduler* in Fig. 3. Furthermore, the MontiBelle approach indirectly increases the cognitive perceptiveness, because instead of proprietary languages, the industry standard language SysML v2 is used. The textual syntax allows adequate structuring by supporting arbitrary formatting. A graphical representation is also offered.

Precision Together with decomposing a system into subsystems and sub-aspects, MontiBelle's stream expressions [Ka22] for specifying requirements allow concisely and separately formulating distinct facts as depicted in List. 1. Besides the signature, there are no additional model artifacts necessary to fully specify systems and their requirements. For close-to-implementation specifications an event-driven, state-based specification technique is provided, which is especially suitable for software-intensive, and time-sensitive systems [KRK13].

Universality MontiBelle suggests strictly history-oriented specifications in the earlier development phases. Besides the system signature and an expected input and output relation, no further assumptions regarding the system's implementation are made. Every implementation that fulfills the input output relation is then conform to the system's specification. Thus, it is ensured that no platform dependencies are introduced. Even in later development phases, MontiBelle recommends abstract (platform independent) state machines as depicted in Fig. 1 and Fig. 2. From these state machines code for any chosen platform that produces output depending on system's states and inputs can be generated by using

a suitable generator. Furthermore, MontiBelle is capable of handling underspecification in models. Underspecification in a specification implies that multiple, potentially infinitely many realizations exist that still fulfill the history- or state-based requirements of the models. Thus, MontiBelle supports variability and management of product-lines.

Simplicity MontiBelle can be actively used to increase simplicity of models. It allows formally verifying semantical compliance of a simpler model (to a previous more complex one), and thus enables correct reformulation and refactoring. Using MontiBelle's underspecification capabilities, refactoring steps permit refinement as well as abstraction of previous models. One example is the refinement and decomposition of the scheduler component into a simpler buffer and capacity component depicted in Fig. 3. Additionally, the approach is at its core syntax agnostic [Ka21b, Ka23]. For domains with assumptions regarding the form of representation of complex properties, a domain-specific adaptation or interchanging the modeling language facilitates domain dependent simplicity (e.g., by requiring a special security keyword to encrypt the transmission, or by using a timing keyword to limit the time delay).

Semantical Adequacy MontiBelle is semantically adequate for developing modern software systems with a focus on safety-critical applications and time-sensitive systems. Since software-systems usually exchange data over a long period of time, the mathematical framework FOCUS is well-suited for depicting such communication histories. The non-starvation requirement of DLUF shows, that liveness properties can be formulated with the MontiBelle approach. Time-critical systems demand specification and analysis of time. For this, timed communication histories and event-driven processing are specially adapted for software-intensive and time-sensitive systems (see [KRK13] for remarks on precision).

Consistency MontiBelle uses the language workbench MontiCore [HKR21] for model processing and consistency analysis. MontiCore offers the possibility to check models for syntax correctness and supports the development of further analyses. Such analyses include checks for compliance with conventions (e.g., naming conventions), reference checks (e.g., existence and visibility of referenced model elements such as system states), and type checks (e.g., type-correct expressions over communication histories, so called stream expressions). The DLUF models are automatically checked in regards to type and interface consistencies.

Conceptual Integrity/Uniformity MontiBelle provides internal conceptual integrity/uniformity through the use of a domain-specific profile for the industry known system modeling language SysML v2. The profile restricts the use of model elements to those that have a semantic foundation in FOCUS, e.g., requirements modeled in List. 1 and state charts modeled in Fig. 1. Of course, it is not sufficient to consider only *internal* conceptual integrity/uniformity. When the subsystems and views of those subsystems come together, the conceptual integrity/uniformity plays a vital role again. The SysML v2 profile therefore is also relevant in Sect. 3.2. Nonetheless, the foundation for this is already laid here.

Conformance MontiBelle requires conformance to the concepts already explained above. Thus, models always consist of the signature of the (sub)system and its behavior, be it descriptive or prescriptive in nature. However, its particular strength lies in the refinement relation between models and the verifiability of those models. For example, MontiBelle achieves conformity of models with respect to EUROCAE ED-216 by external, i.e., cross-model, features (compare Sect. 3.2).

Language-specific, Semantical Quality properties This is one of the core competencies of MontiBelle. By building upon a suitable semantical foundation and mapping models into theorem prover code, MontiBelle achieves semantic analyzability of model artifacts, including abstract, history-oriented specifications and underspecification. This is showcased through the verification of the modeled DLUF system in regards to its non-starvation property. Even complex systems can be formally analyzed by leveraging decomposition in FOCUS and its compatibility with refinement. This verification further enhances model quality.

3.2 Horizontal Inter-model Quality Properties

In the following sections, we address inter-model quality indicators at one granularity level. Granularity levels emerge in the iterative system development process as soon as the system is described at a different level of abstraction. According to the MontiBelle methodology, underspecification is decreased across granularity levels, while decomposition is increased.

Consistency, Conceptual Integrity, Language-specific, Semantical Quality Properties These properties can often be assessed only across models. The MontiBelle SysML profile allows system engineers to model refinement dependencies across model artifacts and the modeled system parts. In DLUF, 19 refinement relations (compare Fig. 5) are traced. These refinement relations between models are statically and semantically verifiable. Static verification is a well established model analysis approach. MontiBelle, for example, checks the compatibility of refined interfaces. These checks are made possible by MontiCore's [HKR21] language infrastructure. Semantic verification, however, is only possible with a suitable semantic domain. MontiBelle uses FOCUS to facilitate the formal verification of horizontal inter-model conformity and conceptual integrity. All refinement relations of DLUF were proven in the case study [Ka22].

Downward Completeness Following the MontiBelle methodology, downward completeness is inherently reached, once the current granularity level's specifications fulfill the requirements from the previous level. This is because MontiBelle is able to verify the correct specification of underspecified systems, as shown multiple times for the DLUF case study granularity levels in Fig. 5. Furthermore, MontiBelle promotes the use of decomposition and refinement to gain more granular and eventually less abstract specifications. Fulfillment of previous specifications can be formally verified, as the paragraph about *correctness* explains. Downward completeness is thus highly related to *correctness* and *upward completeness* in the MontiBelle approach.

Cohesion (De)composition supports both cohesion and modularity of models. According to the MontiBelle methodology, interrelated system aspects are modeled as individual subsystems, because this facilitates verification. In the DLUF models this is exemplary shown in the scheduler component. A strong cohesion between the buffer (see Fig. 1) and capacity models (see Fig. 2) exists, since together they specify the scheduler (see Fig. 3). The scheduler model is individually well-defined and can be understood without any other models. Through the methodical application of the MontiBelle approach, closely related system parts of the DLUF system are cohesively modeled.

Modularity (De)composition (see Fig. 5) is arguably even more important for modularity than it is for cohesion. By decomposing a system into multiple subsystems, the system and its models get more

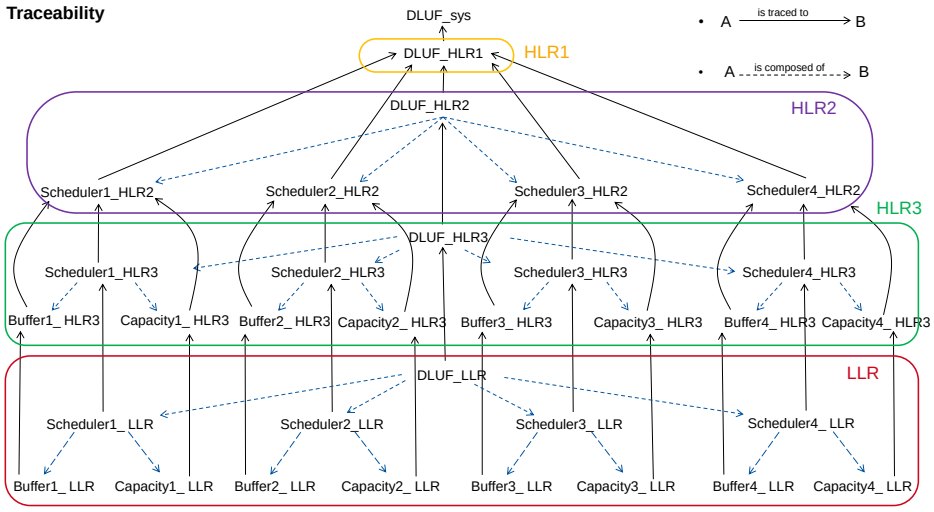


Fig. 5: Hierarchical decomposition of the DLUF case study including tracing and decomposition.

and more modular, since the models of subsystems only model single aspects. This modularity allows independent development into new granularity levels including verifying refinement relations. The independently developed models' interfaces are correct in regard to the system, because the refinement of a subsystem directly implies the refinement of the whole system in Focus. The buffer component in the DLUF system can independently be developed further without having any implication on the scheduler component or the whole DLUF system.

Freedom from Redundancies For systems modeled according to the MontiBelle approach, a granularity level reduces redundancies by importing and re-using models. The scheduler composition in Fig. 3 does not define the buffer's or capacity's interfaces and behavior. Instead, model artifacts are referenced for the composition. Furthermore, additional redundancies are removed in MontiBelle by facilitating parametric models. Only one capacity model is enough to specify the four *different* scheduler components of DLUF [Ka22]. However, completely redundancy free modeling is impossible, e.g., when referencing other model artifacts a model name is used. If this model name is changed, the references to this model must also be changed[5]. Additional well-defined tooling mitigates this problem.

Controlled Redundancy By decomposing the system into individual models, thus introducing controlled redundancy, these models can be independently developed. Model redundancies, when referencing models, allow statically checking type and structural correctness of compositions. Verification coverage is given, since MontiBelle formally verifies the correctness of a granularity level. In DLUF, different system engineers specify and refine different models.

[5] Name changes can automatically be handled by renaming all references. MontiCore [HKR21] provides a symbol infrastructure for this purpose, enabling tracing named references across all model files.

3.3 Vertical Inter-model Quality Properties

In the following sections cross-model and cross-granularity level quality properties are evaluated for the MontiBelle methodology. These quality indicators can be assessed, once model artifacts of systems are iteratively developed.

Correctness For correctness, the requirements from the previous level must have been correctly implemented in the following level. MontiBelle allows the modeling and correctness checking through the refinement relation of the FOCUS theory. This is done in a machine-supported and automated way. For DLUF, all system specifications of all granularity layers and all refinement relations between them are automatically translated into theorem prover syntax. Then all proof obligations are met by automatic solvers and machine checked [Ka22]. In summary, this verified, that the architecture and LLR are correct regarding the top level HLR. MontiBelle not only verifies this quality attribute, but also supports the development of models in this direction. Counterexamples can be found and extracted as test cases, which can facilitate revisions of the models.

Upward Completeness Orthogonal to correctness, this property requires, that all requirements from the level above have been fulfilled. The reasoning therefore follows in large parts that of the previous paragraph. A particularly important feature of MontiBelle's underlying FOCUS theory is the compositionality of refinement. Due to FOCUS, the global consideration of requirements and development steps is reduced to many small, local refinement proofs, which are illustrated in Fig. 5. As a result, the complexity of proof finding is decreased, reusability is stimulated, and engineering productivity can be increased through parallelization.

Traceability MontiBelle ensures traceability through refinement relations. Refinement relations can be continuously checked and thus improve traceability compared to unchecked tracing relations. The continuous verification of the refinement relations is possible locally, for example in an Integrated Development Environment (IDE) like environment. It is also possible to verify the refinement relations in batch mode, for example in GitLab© CI/CD or GitHub© Actions. This is particularly important when multiple subsystems are integrated into an overall architecture. Changing a requirement immediately leads to updated proof obligations. These proof obligations are automatically encoded in a theorem prover via a code generator. The proof obligations can be checked automatically through automation of proof finders and tactics [Bü20]. Furthermore, because traceability is a key indicator for upwards completeness, the same reasoning as to why MontiBelle increases upwards completeness apply here. Particularly, the compositionality of refinements allows us to automatically conclude refinement of complete systems from refinement of individual subsystems as depicted in Fig. 5. Plainly put, this increases the value of those individual refinement relations. By increasing their value and encouraging their methodological usage, MontiBelle increases the traceability.

Modifiability Modifiability can be divided into the three aspects (1) maintainability, (2) extensibility, and (3) reusability [FHR08]. We will argue each aspect individually in the following paragraphs.

MontiBelle improves maintainability by assuring correctness of changes induced by maintenance work to existing requirements. By using MontiBelle, the semantics of the original and the changed specification can be compared. It can thus be formally verified, that the maintenance work had the intended effect on the system specification. Additionally, relations to other system specifications (for

example refinement to hardware- pecific requirements) can continuously be verified, as well. It is important to note, that MontiBelle is able to verify underspecified system specifications, thus can be used to maintain even early and relatively vague system specifications. The verification results can be used to steer the maintenance work. This enables semantic oriented maintenance with formally assured outcome.

Extensibility is improved analogously. With MontiBelle, extended system specifications can be semantically verified regarding old and new requirements or compared to the original system specification. The use of underspecification in early development phases enables proving the correctness of or deliver counterexamples for extended system specifications. As an example, assume a system specification was developed from safety and security constraints. Now assume the system specification is extended in such a way as to save resources. For example, subsystems might be merged to deliver multiple functionalities and thus save costs. Communication channels might be reduced to save cable runs and thus weight. Such optimizations might be unsafe as merging multiple systems might introduce unsafe behavior or reduced cable runs could decrease fault tolerances. For safe systems, MontiBelle is able to provide assurances in the form of formal safety proofs. If the extended system specification is not safe, then MontiBelle can be used to produce and check potential counterexamples. Use of underspecification enables modeling and verification of such optimizations at early development stages.

MontiBelle promotes model reusability by allowing models to be placed in a refinement hierarchy. More abstract requirements can serve as the starting point for a tree of developed system specifications. Each developed system specification might be an independent subset of the specification. This enables the creation (and maintenance, see above) of product families. The hierarchical tree structure allows verification artifacts to be reused. Assume an avionics data transmission system specification was developed from a set of regulatory requirements. An overview of the development artifacts is shown in Fig. 6. The requirements leave some fixed maximum tolerance to the transmission delay. The developed specification must adhere to those tolerances, but is still underspecified regarding the exact delay. A top-of-the-line system was developed from those specifications, having a delay of at most 1 ms. For a less mission critical application, a cheaper system with a higher delay tolerance can now be developed from the same underspecified system specification. With MontiBelle, it suffices to show the correct development of this cheaper system from the intermediary specification. This reduces the overall costs of product families.

Freedom of Redundancies The DLUF system case study includes a buffer HLR [Ka22] and a buffer LLR model. These specify the behavior of a buffer in different abstraction levels. Changing the behavior or interface of one of the models may cause the other model to need to be modified, as well. Thus, the MontiBelle methodology is not free of redundancies. However, this problem is mitigated by specifying the refinement relation. Should redundant information be inconsistent syntactically, MontiBelle provides model analysis tools to find these inconsistencies. If the inconsistencies are of semantical nature, then MontiBelle automatically attempts a re-verification. If this verification is successful, the change is re-verified (see *Modifiability*). If it fails, MontiBelle can provide the system engineer with a counterexample regarding the refinement relation. The redundancy in behavior description is desirable and is therefore not a defect of the methodology.

Controlled Redundancies Controlled redundancies in the MontiBelle approach give the possibility to specify behavior both descriptively and prescriptively [Ka23]. Descriptive specifications, in the form of HLRs, are represented by history-based specifications. Prescriptive specifications (LLRs) are represented as state-based specifications by event automata. These redundancies allow for correctness and consistency checks between the speficiations, ultimately increasing correctness of the

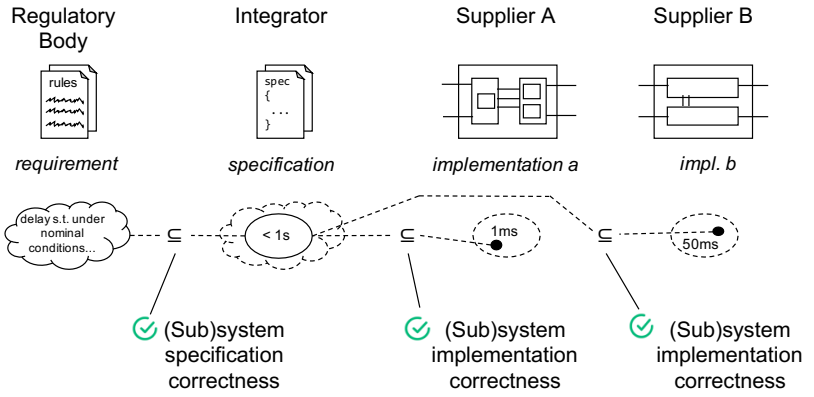


Fig. 6: Exemplary development of an avionics data transmission system with delay constraints. An integrator would have to prove its specifications to be safe regarding some regulatory body's requirements (left). Supplier A would have to prove its system's correctness regarding the integrator's specifications.

developed system. Models can also be provided in a reader-oriented manner by (de)composition. A requirements manager may need a more abstract view of the system and thus considers a descriptive specification of a composed system (e.g., DLUF HLR in List. 1) [Ka22]. A software developer of a specific subsystem meanwhile is more interested in the prescriptive, state-based behavior of the respective subsystem (e.g., buffer LLR in Fig. 1).

3.4 Quality Properties for Modeling Notation

Some model quality indicators go beyond measuring individual models and rather measure the modeling notation itself.

Degree of Formalization Using MontiBelle increases the level of formalization in two ways. First, the SysML v2 profile requires a minimum level of formalization. The interfaces of the systems and subsystems and their interconnections must be specified. It is permitted to adapt the interfaces and structures in later developmemt steps, but one development step is not complete, until the structure of the system specification has been defined at this step and thus granularity level. Further, behavioral specifications cannot be described informally at will, but must be done in one of three ways. These are: history-oriented (abstract), state-oriented (implementation-oriented) and decomposition-oriented (structural). Beyond this minimal degree of formalization, MontiBelle allows arbitrary underspecified behavior, which remains at most the same or is refined with each development step. Thus, after a refinement step the specificity increases and the solution space becomes smaller.

Adequacy for the Application Domain First, MontiBelle is language agnostic in terms of concrete syntax. As shown in [Ka21b, Ka21a], an intermediate representation of models captures the semantic domain of architecture description languages and behavioral specifications. This intermediary

model is then mapped to FOCUS. Thus, a domain coupled with a corresponding domain specific language (DSL) can be easily served, as long as the semantic domain is compatible. As one of the concrete modeling languages realized by MontiBelle, SysML v2 is the successor of the de facto standard language of systems engineering, SysML v1. SysML v1 is widely known, used, and understood worldwide. The successor, SysML v2, offers many enhancements and improvements thanks to its textual representation with the same graphical visualization and is therefore suitable as a language for the domain of systems engineering and was used in the DLUF case study.

3.5 Additional Model Quality Indicators

This list of model quality indicators presented here is not exhaustive. Different domains might come with their own set of quality indicators. We found the two below to be of particular interest for the avionics domain.

Verifiability MontiBelle requires a certain degree of formalization of the models, as described in the corresponding paragraph *degree of formalization*. Through this formalization, one gains verifiability, as it enables machine assisted formal verification. Specifically, MontiBelle maps the model to its FOCUS semantics via an intermediate syntax agnostic model. As a result, definitions and theorems are encoded in a theorem prover and linked to the core definitions of FOCUS, which have also been encoded in the theorem prover. The theorem prover allows reasoning over the semantics of the models. The mathematical underpinning by FOCUS enables capturing underspecification and abstraction. However, MontiBelle not only enables verifiability, but also improves the verifiability through its development methodology. The methodology is based on the SPES methodology and uses granularity layers and decompositon hierarchies in modeling. This enables not only the decomposition of system requirements (which increases reusability), but leads to decomposed proof obligations, as well. This leads to smaller and thus more automatable verification steps. The verification of the overall system is ensured by the compositionality of the refinement in FOCUS. This property holds by construction and can be checked automatically in our Isabelle encoding.

Transformability MontiBelle achieves transformability of models in the following way: First, any data flow and structure modeling language is reduced to semantically well-founded models. In MontiBelle, this is achieved for SysML v2 by a corresponding profile. This profile ensures, that all models that correspond to the profile are transformable, i.e. semantically sound. The transformability then allows the automated and therefore less error-prone translation of the models into a theorem prover. This is detailed in the paragraph about *verifiability*. The textual representation of SysML v2 ensures, that this transformability is given for all SysML v2 models and is independent of any vendor specific tool and data structure of the models. All models of the DLUF case study were transformed and could subsequently be treated by the highest level of formal methods, i.e. deductive reasoning (theorem proving).

4 Conclusion

In summary, modeling along the MontiBelle methodology has a positive impact on all model quality indicators (answering **RQ (02)**) listed in Tab. 1. As argued previously in [FHR08], increasing model quality can also enhance product quality and thus lead to more correct systems. Accordingly, the

methodology is a good candidate for systems engineering, as it allows the verification and validation of system properties in early development phases. This is especially important for the development of highly safety-critical systems of the avionics domain. For this purpose the MontiBelle tool also offers extensive automation to support the development engineers.

5 Outlook

In order to further reduce modeled redundancy, it is conceivable to increase the use of the extensive parameterization possibilities of SysML v2. In particular, SysML provides the possibility to refer to model elements by so-called *references* instead of repeating them. The extent to which, for example, building blocks can be reused in the form of *constraints* is to be evaluated. We also plan to increase the level of automation even more. A possible next step would be to consider the use of the references to make it easier to check decomposed systems for correctness. Crucially, references to the same constraints could simplify the proof of equivalence or implication between constraints. Additionally, the construction of a library of standard model elements with accompanying certification artifacts seems to be an interesting aspect. MontiBelle benefits from the fact, that the underlying formalism FOCUS already follows exactly this idea. Re-use of development artifacts in an avionics development processes could provide opportunities for a more economic development. Integrating the technique of continuous verification introduced in this paper into avionics development processes might be an incremental step towards this goal. Finally, we plan to evaluate our methodology on models of systems in productive development. Some quality indicators are hard to measure and require the measurement of proxy indicators instead. Furthermore, quantitative assertions are hard to prove. These shortcomings should be further investigated.

Bibliography

[AL94] Abadi, Martín; Lamport, Leslie: Open Systems in TLA. In: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing - PODC '94. ACM Press, New York, New York, USA, pp. 81–90, 1994.

[BD02] Bansiya, J.; Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering, 28(1):4–17, 2002.

[Be00] Berry, Gérard; Bouali, Amar; Fornari, Xavier; Ledinot, Emmanuel; Nassor, Eric; de Simone, Robert: ESTEREL: a formal method applied to avionic software development. Science of Computer Programming, 36(1):5 – 25, 2000.

[BKR09] Becker, Steffen; Koziolek, Heiko; Reussner, Ralf: The Palladio Component Model for Model-Driven Performance Prediction. Journal of Systems and Software, 82:3–22, 01 2009.

[Bö21] Böhm, Wolfgang; Broy, Manfred; Klein, Cornel; Pohl, Klaus; Rumpe, Bernhard; Schröck, Sebastian, eds. Model-Based Engineering of Collaborative Embedded Systems. Springer, Cham, January 2021.

[BR07] Broy, Manfred; Rumpe, Bernhard: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik-Spektrum, 30(1):3–18, Februar 2007.

[Br19] Broy, Manfred: Logische und Methodische Grundlagen der Programm- und Systementwicklung. Springer Fachmedien Wiesbaden GmbH, Wiesbaden, Germany, 2019.

[Br23] Broy, Manfred: Logische und Methodische Grundlagen der Entwicklung verteilter Systeme. Springer-Verlag GmbH, Berlin, Germany, 2023.

[BS04] Broy, Manfred; Steinbrüggen, Ralf: Modellbildung in der Informatik. Springer-Verlag, Berlin, Heidelberg, 2004.

[Bü20] Bürger, Jens Christoph; Kausch, Hendrik; Raco, Deni; Ringert, Jan Oliver; Rumpe, Bernhard; Stüber, Sebastian; Wiartalla, Marc: Towards an Isabelle Theory for Distributed, Interactive Systems – The Untimed Case. Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, Germany, March 2020.

[Ca47] Carnab, Rudolf: Meaning and Necessity: A Study in Semantics and Modal Logic. The University of Chicago Press, Chicago, IL, USA, 1947.

[Ca87] Caspi, Paul; Pilaud, Daniel; Halbwachs, Nicolas; Plaice, John: Lustre: A Declarative Language for Programming Synchronous Systems. In: POPL. 1987.

[CM14] Cofer, Darren; Miller, Steven P: Formal methods case studies for DO-333. Technical report, 2014.

[Dz16] Dziwok, Stefan; Pohlmann, Uwe; Piskachev, Goran; Schubert, David; Thiele, Sebastian; Gerking, Christopher: The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling. Technical report, Software Engineering Department, Fraunhofer IEM, Paderborn, Germany, December 2016.

[FHR08] Fieber, Florian; Huhn, Michaela; Rumpe, Bernhard: Modellqualität als Indikator für Softwarequalität: eine Taxonomie. Informatik-Spektrum, 31(5):408–424, Oktober 2008.

[Ge23] SpesML Project Site. `https://spesml.github.io`, Accessed: 2023-10-24.

[Gr08] Grönniger, Hans; Krahn, Holger; Pinkernell, Claas; Rumpe, Bernhard: Modeling Variants of Automotive Systems using Views. In: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen. Informatik Bericht 2008-01. TU Braunschweig, pp. 76–89, 2008.

[HKR21] Hölldobler, Katrin; Kautz, Oliver; Rumpe, Bernhard: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, Düren, May 2021.

[Ho85] Hoare, C. A. R.: Communicating Sequential Processes. Prentice Hall International, Englewood Cliffs, N.J., 1985.

[IE10] IEC: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. Standard, International Electrotechnical Commission, Geneva, 2010.

[Ka20a] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems. In: Combined Proceedings of the Workshops at Software Engineering 2020. CEUR, Online, February 2020.

[Ka20b] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: MontiBelle – Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety. In: AIAA Scitech 2020 Forum. American Institute of Aeronautics and Astronautics, Reston, Virgina, January 2020.

[Ka21a] Kausch, Hendrik; Michael, Judith; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard; Schweiger, Andreas: Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In: Aerospace Europe Conference 2021 (AEC 2021). Council of European Aerospace Societies (CEAS), Warsaw, Poland, November 2021.

[Ka21b] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams. In: Proceedings of the Software Engineering 2021 Satellite Events. CEUR, Online, February 2021.

[Ka22] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard; Schweiger, Andreas: Correct and Sustainable Development Using Model-based Engineering and Formal Methods. In: 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC). IEEE, USA, September 2022.

[Ka23] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rath, Amelie; Rumpe, Bernhard; Schweiger, Andreas: A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System. In: Software Engineering 2023 Workshops. Gesellschaft für Informatik e.V., Bonn, pp. 169–188, February 2023.

[Kr19] Kriebel, Stefan; Raco, Deni; Rumpe, Bernhard; Stüber, Sebastian: Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In: Proceedings of the Workshop on Avionics Systems and Software Engineering (AvioSE'19). CEUR, Online, pp. 87–94, February 2019.

[KRK13] Kounev, Samuel; Rathfelder, Christoph; Klatt, Benjamin: Modeling of Event-based Communication in Component-based Architectures: State-of-the-Art and Future Directions. Electronic Notes in Theoretical Computer Science, 295:3–9, 2013. Proceedings the 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).

[Le97] Lehman, M.M.; Ramil, J.F.; Wernick, P.D.; Perry, D.E.; Turski, W.M.: Metrics and Laws of Software Evolution - The Nineties View. In: Proceedings Fourth International Software Metrics Symposium. pp. 20–32, 1997.

[Le16] Lee, Edward: Fundamental Limits of Cyber-Physical Systems Modeling. ACM Transactions on Cyber-Physical Systems, 1:1–26, 11 2016.

[Mi82] Milner, R.: A Calculus of Communicating Systems. Springer-Verlag, Berlin, Heidelberg, 1982.

[ML09] Murray, Toby; Lowe, Gavin: On Refinement-Closed Security Properties and Nondeterministic Compositions. Electr. Notes Theor. Comput. Sci., 250:49–68, 09 2009.

[NPW02] Nipkow, Tobias; Paulson, Lawrence C.; Wenzel, Markus: Isabelle/HOL: A proof assistant for Higher-Order Logic. Lecture Notes in Artificial Intelligence. Springer, Berlin et al., 2002.

[Pa01] Parrow, Joachim: Handbook of Process Algebra. Elsevier Science, Amsterdam, chapter An Introduction to the Π-Calculus, pp. 479–543, 2001.

[PR94] Paech, Barbara; Rumpe, Bernhard: A new Concept of Refinement used for Behaviour Modelling with Automata. In: Proceedings of the Industrial Benefit of Formal Methods (FME'94). LNCS 873, Springer, Spain, pp. 154–174, 1994.

[PR97] Philipps, Jan; Rumpe, Bernhard: Refinement of Information Flow Architectures. In: ICFEM'97 Proceedings. IEEE CS Press, Hiroshima, Japan, 1997.

[Re85] Reisig, Wolfgang: Petri Nets: An Introduction. Springer, Berlin, Heidelberg, 1985.

[RJH03] Rayadurgam, Sanjai; Joshi, Anjali; Heimdahl, Mats P. E.: Using PVS to Prove Properties of Systems Modelled in a Synchronous Dataflow Language. In: Formal Methods and Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 167–186, 2003.

[St73] Stachowiak, Herbert: Allgemeine Modelltheorie. Springer, 1973.

[Sy23] SysML v2 Official Specification. `https://github.com/Systems-Modeling`, Accessed: 2023-11-29.

[VZ14] Voss, Sebastian; Zverlov, Sergey: Design Space Exploration in AutoFOCUS 3 – An Overview. In: IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems. 2014.

Workshop Generative and Neurosymbolic AI in Software Engineering (GenSE'2024)

Reliable Generation of Formal Specifications using Large Language Models

Philipp Kogler[1], Andreas Falkner[2], Simon Sperl[3]

Abstract: Recent pre-trained Large Language Models (LLMs) have demonstrated promising Natural Language Processing (NLP) and code generation abilities. However, the intrinsically unreliable output due to the probabilistic nature of LLMs imposes a major challenge as validity can generally not be guaranteed, making subsequent processing prone to errors. When LLMs are used to translate natural-language specifications to formal specifications, this limitation becomes evident. We propose a framework involving prompting and algorithmic post-processing that continuously interacts with the LLM to ensure strict syntactic validity and reasonable content correctness. Furthermore, we introduce a use-case in the domain of engineering processes for railway infrastructure and demonstrate that our approach is sufficiently mature for implementation in an industrial environment.

Keywords: Generative AI; Large Language Models; Reliable Code Generation; Post-processing; Domain-specific Languages; Engineering Processes

1 Introduction

Recent advances in AI research related to pre-trained Large Language Models (LLMs) have caused a paradigm shift for numerous tasks in Natural Language Processing (NLP), such as text classification, text generation, text processing, or information retrieval. Traditionally, a latent feature representation must be designed or learned individually for each NLP task, with the quality of the outcome being highly dependent on the training data. Since language understanding is a common factor of many NLP tasks, the idea of pre-training this generic and task-agnostic aspect became more widely adopted [Mi23].

Domain-specific languages (DSLs) are an approach to accelerate software development in specialized domains. In contrast to general-purpose programming languages, DSLs are tailored towards a specific task or problem. Writing DSL code requires knowledge of the DSL itself as well as programming skills. Domain experts often lack these requirements and collaboration with software developers becomes essential. Leveraging the outstanding NLP abilities of LLMs, DSL code can be automatically generated from natural-language specifications written by domains experts, presenting an opportunity to streamline development processes in which DSLs are applied. LLMs have shown reasonable abilities in machine-translation tasks from natural-language descriptions to code [Xu22].

[1] Siemens AG Österreich, Siemensstraße 90, 1210 Wien, Austria philipp.kogler@siemens.com
[2] Siemens AG Österreich, Siemensstraße 90, 1210 Wien, Austria andreas.a.falkner@siemens.com
[3] Siemens AG Österreich, Siemensstraße 90, 1210 Wien, Austria simon.sperl@siemens.com

doi:10.18420/sw2024-ws_10

Two main challenges are imposed by the application of pre-trained LLMs to DSL code generation [Li23b]:

Domain-specific knowledge: Pre-trained LLMs are a general-purpose solution. They lack specific domain knowledge and the concrete syntax of the DSL.

Probabilistic nature: A specific output format or satisfaction of given constraints are generally not guaranteed.

In this paper, we propose a framework to reliably generate DSL code from natural-language specifications using a LLM. Furthermore, we evaluate our solution through a concrete industrial use-case in the domain of engineering processes, and show that our post-processing procedure ensures the satisfaction of validity constraints for the DSL code that is generated by the LLM.

The paper is organized as follows: In Section 2, relevant background regarding LLMs is explained. In Section 3 our framework is described, consisting of prompting, the model, and our post-processor. Furthermore, the use-case of engineering processes in railway infrastructure is introduced in Section 4.1, followed by the adaptation of the generic framework to the use-case in Section 4.2, and extensive evaluations in Section 4.3. The paper is concluded by listing related work (Section 5) and a summary (Section 6).

2 Background

Essential concepts used in our proposed reliable DSL code generation framework are auto-regressive transformer models, domain-specific customization of pre-trained LLMs, and specialized code LLMs.

2.1 Auto-regressive transformers

The use of LLMs for NLP tasks such as text classification, generation or processing was a major paradigm shift. Instead of designing and learning feature representations individually, pre-training the task-agnostic aspect of natural language understanding became widely adopted, especially with the introduction of the *Transformer* architecture that allowed model pre-training on a larger scale combined with further improvements, most notably the efficient consideration of long-range dependencies within texts [Va17].

The *decoder-only* models are a subclass of Transformer-based architectures and always output text. They are used for text continuation or prompting and are most commonly trained auto-regressively. Auto-regressive models predict the next single token (sub-word) by maximizing the log-likelihood given all previous words and the model parameters [Va17]. Two state-of-the-art examples are GPT-4 [Op23] and Llama 2 [To23].

2.2 Domain-specific customization

Pre-trained LLMs are typically trained on a large publicly available text corpus and are a general-purpose solution. Therefore, they are not easily capable to operate in a highly specialized domain as expert knowledge may not be freely available. Domain-specific customization quickly becomes a necessity when applying LLMs to practical tasks and can be achieved by employing techniques like *prompting* and *fine-tuning* [Li23b].

Prompts are task-specific input texts, and *prompt crafting* is the technique of carefully designing prompts to guide the LLM towards a desired output. *Discrete prompting* uses static prompt texts without updating inner model parameters and is categorized as *zero-shot* or *few-shot*. While zero-shot prompts only contain the prompt text itself, few-shot prompts also contain a few concrete examples to specify the expected output [Li23b].

Fine-tuning is the technique of applying additional training on a smaller, more specialized dataset to a pre-trained model to enhance its performance in a specific domain [Li23b].

2.3 Code LLMs

A crucial aspect of LLMs is the pre-training corpus. Increasing its size and diversity improves model performance, if the quality of the training data remains sufficiently high. Recent models are trained on a vast amount of publicly available data from the web [Mi23]. Although general-purpose pre-trained LLMs have shown reasonable performance in completing or synthesizing code from natural language descriptions, specialized models trained on code with a variety of programming languages together with natural-language descriptions are more suitable [Xu22]. Examples of such code LLMs are the proprietary Codex [Ch21] model, a GPT-3 variety that was fine-tuned on code, and Code Llama [Ro23], an open-access code LLM based on Llama 2.

3 Framework for Reliable Code Generation

Our proposed framework, depicted in Figure 1, addresses both challenges, the lack of domain-specific knowledge as well as the probabilistic output generation. The natural-language specification is provided by the domain expert. Together with context, the formal syntax definition of the DSL, and representative DSL code examples, it is handed over to the LLM. The post-processor continuously interacts with the LLM in each generation step and strictly guides it towards generating valid DSL code. The result is a valid DSL code instance.

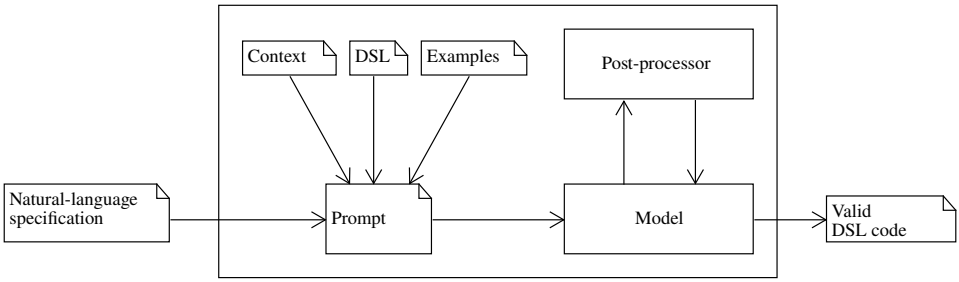


Fig. 1: Framework

3.1 Prompt

Knowledge of the domain is introduced through a combination of three well-known prompting techniques:

Context. General pre-defined instructions for the LLM [NLL23] describe the translation task from natural language to DSL code and tell the LLM to comply with the context and user input.

DSL. The formal specification of the DSL itself is provided as an input to the LLM. Each language element and property is enriched by a natural-language description to enable the LLM to map the contents of the natural-language description provided by the domain expert to appropriate DSL constructs.

Examples. A small set of diverse DSL code instances is attached as pairs of natural-language process descriptions and corresponding code (few-shot prompting). It has been shown that with advanced prompting techniques, in particular similarity-based selection of few-shot examples from a larger example pool together with schema and context information, performance similar to fine-tuning can be achieved in specific domains [Na23]. Effective fine-tuning is dependent on the amount and quality of available datasets [Bu23], and was not considered due to the need for a large amount of training data.

3.2 Model

Our framework relies on open-access auto-regressive transformer models from *Hugging Face*, a machine learning platform where many models are published by the (research) community. While the current implementation uses the *Hugging Face Transformers API*, the general concept is not limited to a specific technology. Regarding the concrete model choice, requirements imposed by the DSL, performance and quality considerations need to be taken into account. The pre-training corpus of the concrete model determines its basic ability to understand the task at hand. Therefore, the pre-training corpus should include programming

languages that are similar to the DSL, and natural-language descriptions in the language used by the domain experts. For instance, if the DSL is JSON-based, the quality of the generated DSL code will increase, if the model was additionally trained on JSON files.

As models become larger (number of parameters), the quality of the generated code usually improves [Xu22]. However, larger models require more processing power, and the speed of model inference (generation of DSL code) depends on model architecture, model size, and hardware, requiring an appropriate trade-off between these factors.

3.3 Post-processor

Our post-processing algorithm, depicted in Figure 2, controls the output generation of the LLM. The auto-regressive transformer model generates its output step-by-step as tokens (sub-words). The post-processor engages into every generation step: For each step, the model generates hundreds or thousands of candidates for the next token based on the prompt and the generated output so far. Sorted by priority as evaluated by the LLM, the post-processor determines whether the token candidate represents a valid continuation of the partial output sequence (partial DSL code) according to the given DSL specification. The valid token candidate with the highest priority is then selected, handed back to the LLM, and added to the partial DSL code, extending it one step further towards a full and valid DSL code instance. A stop criterion is employed and evaluated after every step to determine, if the DSL code instance is complete according to the DSL specification.

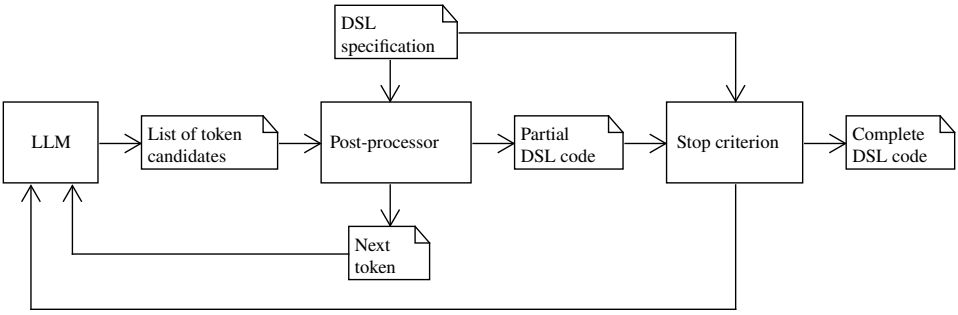


Fig. 2: Post-processing

4 Use-case & Evaluation

We evaluate our framework for reliable DSL code generation on an industrial use-case in the domain of engineering processes for railway infrastructure.

4.1 Introduction to SHAPE Engineering Processes

Engineering processes consist of a set of sequential or parallel tasks with greatly varying diversity and complexity. Common approaches to describe and implement such processes are graphical notations or domain-specific languages. A leading general-purpose solution is the Business Process Modeling Notation (BPMN) [CT12].

Industrial applications introduce highly specific requirements. The implementation of engineering processes in railway infrastructure aims to guide engineers through rigorous, extensive and safety-critical processes to optimize their execution and ensure compliance with safety rules in the domain. For example, appropriate resources must be assigned to tasks, work must be traceable and verifiable for future audits, and documentation requirements must be satisfied [Ba16].

The SHAPE[4] application framework enables the specification of such processes through a flexible Java API. The application then executes the specified processes, collects and monitors data from diverse data sources, continuously evaluates data consistency and completeness through constraints, automates repeated tasks, creates virtual documents, and more. Application users, typically engineers or managers, access the graphical user interface via a standard web browser.

A simplified domain-specific language (SHAPE-DSL) was defined as an abstraction over the SHAPE Java API and enables process designers to model the structure, data, and input forms of SHAPE engineering processes. Technologically, the SHAPE-DSL is JSON[5]-based and a JSON schema defines its syntax. The core language elements of the SHAPE-DSL are listed below, and appendix 1 shows an example of a SHAPE-DSL code instance.

Process. As the overarching structure, the process serves as a container for the set of tasks and general metadata.

Task. A task is a work unit that, during the execution of the process, is assigned to a person and has a state (created, started, or finished). Depending on the state, dynamic forms are displayed. Tasks can consist of multiple sub-tasks.

Form. Forms contain persistent data items (string, number, boolean, date, file, or user) and corresponding form fields.

Field. User inputs are collected through different types of fields (standard text input fields, dropdowns, file selectors, buttons, tables) that contain a user-facing label and description as well as other field-related metadata.

Constraint. Data validations on fields are defined through constraints.

[4] Safety-critical Human- and Data-centric Process Management in Engineering Projects
[5] JavaScript Object Notation

4.2 Post-processor for the SHAPE-DSL

To reliably generate valid SHAPE-DSL code, our framework requires a specialized post-processor (see Section 3.3). In the context of the SHAPE-DSL, a partial JSON object needs to be validated against a JSON schema. General-purpose libraries that validate JSON objects against a schema are limited to validating complete JSON objects. However, given any generation state, i.e., any partial SHAPE-DSL code, our post-processing algorithm evaluates whether a generated token is a valid continuation. To achieve this, our streaming JSON validator is able to strictly validate any such partial JSON object against the SHAPE-DSL JSON schema. Each generic JSON language element (object, list, string, number, etc.) is represented by a deterministic finite automaton (DFA), keeping track of the current state. The token generated by the LLM is broken down to single-character inputs for the JSON validator. Depending on the dynamically parsed JSON schema and the current state, only a set of characters is accepted. If a character is rejected, the current token is considered invalid, and the validator state is rolled back to the last valid token. State changes are triggered by characters until the final state is reached. When the DFA is finished, the generated valid SHAPE-DSL code is complete, and the LLM is signaled to stop.

4.3 Evaluation

Since the generated result is DSL code, the evaluation was done through automated unit tests because other similarity-based comparison methods commonly used to evaluate text are less suitable for code [Ch21]. Our hand-crafted dataset of 54 diverse test cases comprises two categories: *create* (creation of a new engineering process from scratch) and *modify* (modification of an existing engineering process in given DSL code). To ensure reasonable variety, five basic test classes (*structure*, *forms*, *fields*, *properties*, and *combined*) were defined, each consisting of between three and nine single test cases. The *combined* test class contains more complex instances with multiple tasks and sub-tasks in up to three levels, forms, and different types of fields. Two additional test classes (*german* and *wording*) test the german language understanding and the dependence on specific wording or text structure. The *wording* test class uses less structured natural-language input and less common synonyms for certain keywords, e.g., 'part' or 'phase', instead of the well-known keyword 'task'.

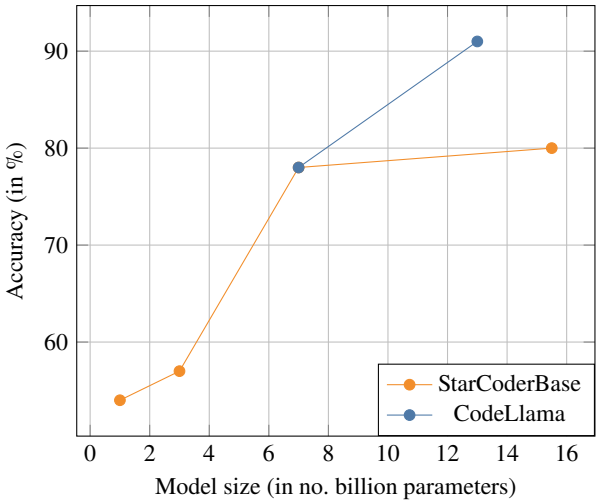
Each unit test consists of a natural-language description, and specific content assertions that are imposed by the natural-language description. We evaluate syntactic and semantic correctness. Syntactic correctness is given, if the generated SHAPE-DSL code passes the validation against the SHAPE-DSL JSON schema. Semantic correctness is given, if the generated SHAPE-DSL code is syntactically correct, and the content of the generated DSL code is correct, i.e., if all content assertions in the concrete unit test are passed.

We evaluated our framework on generating SHAPE-DSL code with two code models, StarCoderBase [Li23a] and CodeLlama [Ro23]. Both showed state-of-the-art performance

on common benchmarks, and were trained on multiple programming languages, including basic JSON syntax. The models are available in multiple sizes. All 54 test cases were evaluated for each model/size combination.

Summarized evaluation results are depicted in Figure 3, plotted as accuracy (percentage of passed test cases) against model size (amount of model parameters). We demonstrated that an increase of model parameters yields better results. We also showed that CodeLlama 13B performs best in our scenario among the models we tested with an accuracy of 91%.

Fig. 3: Evaluation results per model size



Detailed evaluation results per test class of the two best-performing models and with/without post-processing are listed in Table 1. Especially noteworthy is the consistent syntactic validity of 100% when post-processing is enabled. However, semantic correctness still highly depends on the concrete model, particularly in more complex instances. For example, StarCoderBase is unable to process less structured textual descriptions (test class *Wording*) while CodeLlama reaches up to 60% accuracy in this test class. CodeLlama also performs better with German natural-language descriptions. An observation that needs to be further investigated is the *Create/Combined* test class for CodeLlama: While the evaluated result achieves a semantic validity of 100% without post-processing, it falls to 75% with post-processing enabled. An explanation for this performance degradation is that the concrete post-processing implementation uses constraint checks that are stricter than necessary, and wrongly identifies a syntactically correct solution that would be also semantically valid as incorrect. Improvements to the implementation could mitigate this effect.

Tab. 1: Evaluation results per test class, model, and post-processing.
S = Syntactic validity, C = Content/Semantic validity.

Model >		StarCoderBase 15.5B				CodeLlama 13B			
Post-processing >		No		Yes		No		Yes	
Test class [#cases]		S [%]	C [%]	S [%]	C [%]	S [%]	C [%]	S [%]	C [%]
Create (35)	Properties (5)	100	60	100	80	100	80	100	100
	Structure (4)	100	100	100	100	100	100	100	100
	Forms (3)	100	100	100	100	100	100	100	100
	Fields (9)	100	89	100	100	100	89	100	100
	Combined (4)	100	75	100	75	100	100	100	75
	German (5)	100	40	100	40	40	40	100	100
	Wording (5)	80	0	100	0	100	40	100	60
Modify (19)	Properties (5)	100	100	100	100	100	100	100	100
	Structure (3)	100	100	100	100	100	100	100	100
	Forms (3)	100	100	100	100	100	100	100	100
	Fields (5)	100	80	100	80	100	80	100	80
	Combined (3)	100	100	100	100	100	100	100	100

5 Related Work

Text-to-SQL. Translating natural language questions to SQL showed up to 67% accuracy using the Codex model (GPT-3) without any additional domain-specific training. Although Codex provides a strong baseline for direct prompting, providing sample data in the prompt greatly improves the accuracy. A critical aspect is prompt design. Without priming the model with context including schema and content examples, the accuracy dropped below 10% [RLB22].

Text-to-Automation. Definitions of process automations were generated through intermediary Constrained Natural Language (CNL) definitions from natural language inputs using a fine-tuning approach with 100 training samples and prompting techniques. Fine-tuning yielded better results [De22].

nl2spec. In the translation of natural language to temporal logic, the framework *nl2spec* achieved strong results of up to 86% accuracy. The central concept is the automatic identification of sub-translations using OpenAI's Codex model followed by interactive modifications by the user through a graphical interface [Co23].

Synchromesh. Using a few-shot prompting technique, semantically similar examples are selected from a larger pool for a given natural language prompt via a similarity metric named *Target Similarity Tuning*. Constraints are enforced through *Constrained Semantic Decoding* to verify syntax validity, scoping or type checks. During the token-by-token construction of the LLM output, a *Completion Engine* provides all valid tokens that can further extend a partial program towards a full correct program [Po22].

6 Summary, Limitations, and Future Work

Enabling domain experts without technical knowledge in software development to write specifications in natural language that are then machine-translated to DSL code has the potential to accelerate development as needs for individual programming and excessive communication decrease. Reasonable performance with 100% syntactic and up to 91% semantic validity was demonstrated in our use-case evaluation. Therefore, the approach presented in this paper is promising to achieve a no-code or low-code goal.

The presented framework ensures syntactic validity, making DSL code generation with LLMs more reliable. However, the translation of the content and intention from natural language to DSL code can currently not be constrained generically with custom rules beyond the implementation of the post-processor. Also, the quality of the translation highly depends on the concrete LLM. Scalability to more complex real-world instances needs to be assessed further. Individual programming remains essential for highly specific requirements that are not easily covered by a simplified DSL.

Future extensions will be: (i) introduce further advanced techniques to the framework; (ii) expand testing to verify that this approach scales to larger, more complex instances; (iii) evaluate the performance of more LLMs; (iv) conduct practical user evaluations with domain experts.

Bibliography

[Ba16] Bala, Saimir; Havur, Giray; Sperl, Simon; Steyskal, Simon; Haselböck, Alois; Mendling, Jan; Polleres, Axel: SHAPEworks: A BPMS Extension for Complex Process Management. In (Azevedo, Leonardo; Cabanillas, Cristina, eds): Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Brazil, September 21, 2016. volume 1789 of CEUR Workshop Proceedings. CEUR-WS.org, pp. 50–55, 2016.

[Bu23] Busch, Kiran; Rochlitzer, Alexander; Sola, Diana; Leopold, Henrik: Just Tell Me: Prompt Engineering in Business Process Management. In (van der Aa, Han; Bork, Dominik; Proper, Henderik A.; Schmidt, Rainer, eds): Enterprise, Business-Process and Information Systems Modeling. Springer Nature Switzerland, Cham, pp. 3–11, 2023.

[Ch21] Chen, Mark; Tworek, Jerry; Jun, Heewoo; Yuan, Qiming; Ponde, Henrique; Kaplan, Jared; Edwards, Harrison; Burda, Yura; Joseph, Nicholas; Brockman, Greg; Ray, Alex; Puri, Raul; Krueger, Gretchen; Petrov, Michael; Khlaaf, Heidy; Sastry, Girish; Mishkin, Pamela; Chan, Brooke; Gray, Scott; Ryder, Nick; Pavlov, Mikhail; Power, Alethea; Kaiser, Lukasz; Bavarian, Mohammad; Winter, Clemens; Tillet, Philippe; Such, Felipe Petroski; Cummings, David W.; Plappert, Matthias; Chantzis, Fotios; Barnes, Elizabeth; Herbert-Voss, Ariel; Guss, William H.; Nichol, Alex; Babuschkin, Igor; Balaji, S. Arun; Jain, Shantanu; Carr, Andrew; Leike, Jan; Achiam, Joshua; Misra, Vedant; Morikawa, Evan; Radford, Alec; Knight, Matthew M.; Brundage, Miles; Murati, Mira; Mayer, Katie; Welinder, Peter; McGrew, Bob; Amodei, Dario; McCandlish, Sam; Sutskever, Ilya; Zaremba, Wojciech: Evaluating Large Language Models Trained on Code. ArXiv, abs/2107.03374, 2021.

[Co23] Cosler, Matthias; Hahn, Christopher; Mendoza, Daniel; Schmitt, Frederik; Trippel, Caroline: nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In (Enea, Constantin; Lal, Akash, eds): Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. volume 13965 of Lecture Notes in Computer Science. Springer, pp. 383–396, 2023.

[CT12] Chinosi, Michele; Trombetta, Alberto: BPMN: An introduction to the standard. Computer Standards and Interfaces, 34(1):124–134, 2012.

[De22] Desmond, Michael; Duesterwald, Evelyn; Isahagian, Vatche; Muthusamy, Vinod: A No-Code Low-Code Paradigm for Authoring Business Automations Using Natural Language. CoRR, 2022.

[Li23a] Li, Raymond; Allal, Loubna Ben; Zi, Yangtian; Muennighoff, Niklas; Kocetkov, Denis; Mou, Chenghao; Marone, Marc; Akiki, Christopher; Li, Jia; Chim, Jenny; Liu, Qian; Zheltonozhskii, Evgenii; Zhuo, Terry Yue; Wang, Thomas; Dehaene, Olivier; Davaadorj, Mishig; Lamy-Poirier, Joel; Monteiro, João; Shliazhko, Oleh; Gontier, Nicolas; Meade, Nicholas; Zebaze, Armel; Yee, Ming-Ho; Umapathi, Logesh Kumar; Zhu, Jian; Lipkin, Benjamin; Oblokulov, Muhtasham; Wang, Zhiruo; V, Rudra Murthy; Stillerman, Jason; Patel, Siva Sankalp; Abulkhanov, Dmitry; Zocca, Marco; Dey, Manan; Zhang, Zhihan; Moustafa-Fahmy, Nour; Bhattacharyya, Urvashi; Yu, Wenhao; Singh, Swayam; Luccioni, Sasha; Villegas, Paulo; Kunakov, Maxim; Zhdanov, Fedor; Romero, Manuel; Lee, Tony; Timor, Nadav; Ding, Jennifer; Schlesinger, Claire; Schoelkopf, Hailey; Ebert, Jan; Dao, Tri; Mishra, Mayank; Gu, Alex; Robinson, Jennifer; Anderson, Carolyn Jane; Dolan-Gavitt, Brendan; Contractor, Danish; Reddy, Siva; Fried, Daniel; Bahdanau, Dzmitry; Jernite, Yacine; Ferrandis, Carlos Muñoz; Hughes, Sean; Wolf, Thomas; Guha, Arjun; von Werra, Leandro; de Vries, Harm: StarCoder: may the source be with you! CoRR, 2023.

[Li23b] Ling, Chen; Zhao, Xujiang; Lu, Jiaying; Deng, Chengyuan; Zheng, Can; Wang, Junxiang; Chowdhury, Tanmoy; Li, Yun; Cui, Hejie; Zhang, Xuchao; Zhao, Tianjiao; Panalkar, Amit; Cheng, Wei; Wang, Haoyu; Liu, Yanchi; Chen, Zhengzhang; Chen, Haifeng; White, Chris; Gu, Quanquan; Yang, Carl; Zhao, Liang: Beyond One-Model-Fits-All: A Survey of Domain Specialization for Large Language Models. CoRR, 2023.

[Mi23] Min, Bonan; Ross, Hayley; Sulem, Elior; Veyseh, Amir Pouran Ben; Nguyen, Thien Huu; Sainz, Oscar; Agirre, Eneko; Heintz, Ilana; Roth, Dan: Recent Advances in Natural Language Processing via Large Pre-Trained Language Models: A Survey. ACM Computing Surveys, 6 2023.

[Na23] Nan, Linyong; Zhao, Yilun; Zou, Weijin; Ri, Narutatsu; Tae, Jaesung; Zhang, Ellen; Cohan, Arman; Radev, Dragomir: Enhancing Few-shot Text-to-SQL Capabilities of Large Language Models: A Study on Prompt Design Strategies. CoRR, 2023.

[NLL23] Ni, Xuanfan; Li, Piji; Li, Huayang: Unified Text Structuralization with Instruction-tuned Language Models. CoRR, abs/2303.14956, 2023.

[Op23] OpenAI: GPT-4 Technical Report, 2023.

[Po22] Poesia, Gabriel; Polozov, Alex; Le, Vu; Tiwari, Ashish; Soares, Gustavo; Meek, Christopher; Gulwani, Sumit: Synchromesh: Reliable Code Generation from Pre-trained Language Models. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022. OpenReview.net, 2022.

[RLB22] Rajkumar, Nitarshan; Li, Raymond; Bahdanau, Dzmitry: Evaluating the Text-to-SQL Capabilities of Large Language Models. ArXiv, abs/2204.00498, 3 2022.

[Ro23] Rozière, Baptiste; Gehring, Jonas; Gloeckle, Fabian; Sootla, Sten; Gat, Itai; Tan, Ellen; Adi, Yossi; Liu, Jingyu; Remez, Tal; Rapin, Jérémy; Kozhevnikov, Artyom; Evtimov, Ivan; Bitton, Joanna; Bhatt, Manish; Ferrer, Cristian Canton; Grattafiori, Aaron; Xiong, Wenhan; Défossez, Alexandre; Copet, Jade; Azhar, Faisal; Touvron, Hugo; Martin, Louis; Usunier, Nicolas; Scialom, Thomas; Synnaeve, Gabriel; Ai, Meta: Code Llama: Open Foundation Models for Code, 2023.

[To23] Touvron, Hugo; Martin, Louis; Stone, Kevin R.; Albert, Peter; Almahairi, Amjad; Babaei, Yasmine; Bashlykov, Nikolay; Batra, Soumya; Bhargava, Prajjwal; Bhosale, Shruti; Bikel, Daniel M.; Blecher, Lukas; Ferrer, Cristian Cantón; Chen, Moya; Cucurull, Guillem; Esiobu, David; Fernandes, Jude; Fu, Jeremy; Fu, Wenyin; Fuller, Brian; Gao, Cynthia; Goswami, Vedanuj; Goyal, Naman; Hartshorn, Anthony S.; Hosseini, Saghar; Hou, Rui; Inan, Hakan; Kardas, Marcin; Kerkez, Viktor; Khabsa, Madian; Kloumann, Isabel M.; Korenev, A. V.; Koura, Punit Singh; Lachaux, Marie-Anne; Lavril, Thibaut; Lee, Jenya; Liskovich, Diana; Lu, Yinghai; Mao, Yuning; Martinet, Xavier; Mihaylov, Todor; Mishra, Pushkar; Molybog, Igor; Nie, Yixin; Poulton, Andrew; Reizenstein, Jeremy; Rungta, Rashi; Saladi, Kalyan; Schelten, Alan; Silva, Ruan; Smith, Eric Michael; Subramanian, R.; Tan, Xia; Tang, Binh; Taylor, Ross; Williams, Adina; Kuan, Jian Xiang; Xu, Puxin; Yan, Zhengxu; Zarov, Iliyan; Zhang, Yuchen; Fan, Angela; Kambadur, Melanie; Narang, Sharan; Rodriguez, Aurelien; Stojnic, Robert; Edunov, Sergey; Scialom, Thomas: Llama 2: Open Foundation and Fine-Tuned Chat Models. ArXiv, abs/2307.09288, 2023.

[Va17] Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Ł ukasz; Polosukhin, Illia: Attention is All you Need. In (Guyon, I.; Luxburg, U. Von; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R., eds): Advances in Neural Information Processing Systems. volume 30. Curran Associates, Inc., 2017.

[Xu22] Xu, Frank F.; Alon, Uri; Neubig, Graham; Hellendoorn, Vincent Josua: A systematic evaluation of large language models of code. Association for Computing Machinery (ACM), pp. 1–10, 6 2022.

Appendix 1: Example of a SHAPE-DSL code instance

```
{
 "name": "Austria Rail",
 "version": 1,
 "roles": ["Process Engineer", "Reviewer"],
 "rootTask": {
  "name": "at_rail",
  "label": "Austria Rail",
  "type": "root-process-task",
  "hasReadAccess": ["Process Engineer", "Reviewer"],
  "hasWriteAccess": ["Process Engineer"],
  "children": [
   {
    "name": "staff",
    "label": "Project Staff",
    "type": "sub-process-task",
    "description": "Assemble a team",
    "hasReadAccess": ["Process Engineer", "Reviewer"],
    "hasWriteAccess": ["Process Engineer"]
   },
   {
    "name": "planning",
    "label": "Planning",
    "type": "sub-process-task",
    "description": "Collect planning data",
    "hasReadAccess": ["Process Engineer", "Reviewer"],
    "hasWriteAccess": ["Process Engineer"],
    "children": [
     {
      "name": "contract",
      "label": "Contract Data",
      "type": "sub-process-task",
      "description": "",
      "hasReadAccess": ["Process Engineer", "Reviewer"],
      "hasWriteAccess": ["Process Engineer"]
     },
     {
      "name": "check",
      "label": "Check",
      "type": "data-task",
      "description": "Verify planning data",
      "hasReadAccess": ["Process Engineer", "Reviewer"],
      "hasWriteAccess": ["Process Engineer"],
      "forms": [
       {
        "name": "created",
        "data": [
         {
          "name": "check_done",
          "type": "bool"
         },
         {
          "name": "check_doc_id",
          "type": "long"
         },
         {
          "name": "check_date",
          "type": "date"
         }
        ],
        "fields": [
         {
          "name": "check_done",
          "label": "Check done by commercial officers",
          "type": "bool"
         },
         {
          "name": "check_doc_id",
          "label": "ID of the check document",
          "type": "long",
          "constraints": [
           {
            "level": "error",
            "text": "required"
           }
          ]
         },
         {
          "name": "check_date",
          "label": "Date of the check",
          "type": "date",
          "constraints": [
           {
            "level": "error",
            "text": "required"
           }
          ]
         }
        ]
       }
      ]
     }
    ]
   },
   {
    "name": "engineering",
    "label": "Engineering",
    "type": "sub-process-task",
    "description": "",
    "hasReadAccess": ["Process Engineer", "Reviewer"],
    "hasWriteAccess": ["Process Engineer"]
   },
   {
    "name": "verification",
    "label": "Verification",
    "type": "sub-process-task",
    "description": "",
    "hasReadAccess": ["Process Engineer", "Reviewer"],
    "hasWriteAccess": ["Process Engineer"]
   }
  ]
 }
}
```

6th Workshop on Software Engineering for Cyber-Physical Production Systems (SECPPS'24)

Modularization Guidelines to Support Control Software Variability in IEC 61499

Shubham Sharma[1], Anna-Lena Hager, Alois Zoitl

Abstract: In the field of Cyber-Physical Production System (CPPS), a substantial number of control software components are integrated with legacy software. This legacy control software has existed in industries for decades and faces maintenance problems due to sub-optimal tool support. Consequently, rigid software structures have emerged, making maintenance difficult and necessitating better support for managing variability. These legacy systems contain an enormous volume of control software, making it impractical to transform manually in terms of variability management. Additionally, there is a growing demand for variability to accommodate to customer-specific requirements. Control software must be flexible and modular enough to fulfill diverse project-specific needs. Guidelines are required to assist control system engineers in determining which control software components must be refactored and which can be left unchanged. We investigated legacy control software from industrial use cases, identified problem areas, and gathered lessons learned. These lessons have been translated into guidelines for future control software modularization. Hence, in this paper, we present a set of guidelines aimed at modularizing IEC 61499 control software, specifically focusing on enhancing control software variability for variability-intensive CPPS.

Keywords: Modularization; Control-Software; Guidelines; Variability

1 Introduction

Legacy software in the Cyber-Physical Production System (CPPS) domain faces multiple problems due to its large volume, rigid control software structures, and non-optimal maintenance support. The extensive volume of legacy software is the result of decades of continuous control software development and its components in the industry. It is a demanding and time-consuming task to maintain this body of control software. The task becomes even more complex when variability is taken into account. As a result of variability, additional dimensions must be considered, such as customization, adaptation, and possible deviations in behavior or requirements between different software instances or variants. These considerations entail additional requirements that may pose practical challenges. On the one hand, there is a shortage of tools to handle variability factors in the maintenance process. On the other hand, there is also an increased workload, such as the need for in-depth expertise in the field of variability management and extensive training on interdisciplinary project management.

[1] Johannes Kepler University, LIT Cyber-Physical Systems Lab, Altenberger Str. 69, 4040 Linz, Austria
{shubham.sharma,anna-lena.hager,alois.zoitl}@jku.at

doi:10.18420/sw2024-ws_11

In parallel, other industries have made progress in adopting software product lines to manage variability management [Be20]. For instance, the automotive domain has successfully created flexible products, reducing time to market and increasing component reuse. Due to the sub-optimal tool support and the lack of specialized skills in variability management, the control software often remains unaltered from the perspective of variability management in the CPPS domain. However, industries in the CPPS domain are similarly motivated to master variability, seeking to decrease overall development and maintenance costs [RZ21].

The predominant approach to improving variability management in the CPPS domain is the development and utilization of visualization techniques [So22]. Software visualization gives developers a new perspective, enabling them to visualize large-scale software systems, including legacy systems.

Tackling the granularity problem in the legacy software is the other approach. The service and system domains have already analyzed the effect of granularity and developed strategies. For instance, [DB15] investigated the effect of the system's granularity on the system's modularity and found a positive correlation. Similarly, [Ch11] tackled the granularity problem by extending the framework for service modularization.

Guidelines are essential to assist control system engineers in determining which control software components must be refactored and which can be left unchanged. If problem areas in the software are identified manually or with visualization tools, guidelines can also serve as a means to assess the quality of the identified issues. Guidelines find application in various domains to cater to different requirements. For example, [LYL14] introduced five guidelines in function block programming, emphasizing dependable programming. These guidelines aimed to eliminate ambiguity and incorrect usage of Function Blocks (FBs), contributing to a reduction in program faults in safety-critical systems. Legacy software also contains sub-optimal structures and patterns that best suit the conventional tool support. [So21] identified these sub-optimal structures that can result in bad modularization and provided a catalog of bad smells. Thus, modularization guidelines with a dedicated focus on variability management are needed. The needed guidelines for CPPS can also incorporate lessons from other works in the FB programming domain. For instance, insights from legacy software visualization, experiences in dependable programming within the reactor domain, and identifying bad smells in legacy software can all contribute valuable perspectives to the development of comprehensive and effective guidelines.

In this paper, we present modularization guidelines for control software variability in the IEC 61499 domain, derived from an industrial use case. By overcoming challenges in modularizing legacy software — addressing issues like comprehensibility and scattered functionalities — we have gained valuable insights that form the base of our proposed guidelines. Thus, these guidelines offer practical improvement steps for managing legacy software, grounded in real-life scenarios, aiming to drive progress in software development and address industry-specific problems effectively.

Furthermore, these guidelines are versatile and can be applied individually or in combination, depending on the specific aspects relevant to the control software. Moreover, the principles outlined are not confined to the IEC 61499 domain and can also be applied in other domains, such as function block programming in IEC 61131-3.

We structured the paper as follows: In Section 2, we delve into the background and explore related work on control software modularization within the IEC 61499 standard. Then, we present the modularization guidelines in Section 3. We conclude our paper and outline future work in Section 4.

2 Background & Related Work

Modularity is the structuring principle that reduces complexity, enhances clarity, provides flexibility, and allows organizational advantages [ME98]. It acts as a mechanism to balance standardization and customization, concepts that are often regarded as opposing [LM96].

2.1 Control Software Encapsulation Level

The control software in IEC 61499 can be expressed with the different FB types that offer different encapsulation levels [Zo14]. These FB types provide users the flexibility to design control software functionalities with varying granularity and complexity of algorithms within the FB. For instance, a *Simple* FB is constrained by the standard to one algorithm, while a *Basic* FB can accommodate multiple algorithms. Additionally, multiple instances of FBs can be encapsulated within a *Subapplication* or a *Composite* FB to define functionality. *Subapplication* can be *Typed* after encapsulation and stored in the type library for future reuse. Encapsulating process sub-functionalities into *Composite* FBs or *Typed Subapplications* allows users to set variation points based on different process functionalities. Users can also determine the hierarchy level based on encapsulation requirements.

2.2 Control Software Modularization

Earlier work on control software modularization in the IEC 61499 domain is based on code smells [So21]. The objective was to minimize one of the code smells, *Feature Envy*, in IEC 61499 control software. Another exploration involved the utilization of graph clustering algorithms to re-modularize IEC 61499 control software [Ba23]. The authors analyzed existing software architectures and applied semantic clustering in an attempt to enhance modularity. In this approach, the authors took the application and clustered control software sub-components with different levels of modularity. This re-modularization initiative successfully reduced the *Feature Envy* factor and improved the overall coupling between modules. It is worth noting that they did not consider the evolution factor and the

effect of re-modularization on different control software variants. With the growing number of system variants, it is essential to consider how this may affect existing modules that have already been re-modularized.

2.3 Functionality Granularity

The functionality granularity expressed in FBs plays an important role in control software expression. For instance, the same functionality can be expressed by using multiple instances of *Simple* FB which support only one algorithm or with a single *Basic* FBs with multiple algorithm support. Refactoring the FB network involves considering how the FBs are combined and the functionality of the individual composed FB. Within [Pa18], the author refactored the function block application and reported the impact on software metrics before and after refactoring. The authors observed an increase in FBs after refactoring, but these FBs had lower vocabulary, shorter length, and reduced complexity compared to the original system. While the understandability of the refactored FB improved, it also led to an overall increase in maintenance due to the higher number of FBs. Additionally, refactoring increased the depth of encapsulation, indicating higher design complexity.

Similarly, the control software of two process variants was modularized [Sh23b]. The work followed the three basic drivers behind modularization: creating variety for customization, utilizing similarity to gain rationalization benefits, and reducing complexity to improve handling [LM96]. These drivers were applied to the two control software variants, which led to the identification of variation points and the creation of variant modules. The variant modules comprise of the granular sub-functionalities in both control software variants and are consistently used in a similar manner. Subsequently, these granular sub-functionalities were encapsulated inside the subapplication, provided with meaningful interface names, and stored as a type in the type library. This module creation established variation points in the control software based on process functionality and reduced the overall complexity of the application.

The authors of [Sh23a] identified the core components and proposed a modularization control software design for mechatronic variants. The approach involved transforming the conventional clone-and-own design into a *Standardized Core Functionality (SCF)* design, comprising four steps. As part of the *SCF* design, the author encapsulated the core process functionality within a FB and named the interface elements. The choice of using a FB of type *Basic* was made, considering the need for multiple algorithms. The core interface elements were assigned meaningful names based on their interaction with mechatronic sub-components. Furthermore, the functionality of diverse mechatronic sub-components was expressed within the subapplication. While the core functionality could have been expressed with multiple instances of *Simple* FBs, the author chose a *Basic* FB with multiple algorithms. This decision aimed to maintain the core functionality, keeping the functionality encapsulated within a single FB.

The various works highlighted in this section focused on different attributes of control software within the IEC 61499 domain, addressing: (1) re-modularization to enhance cohesion between FBs, (2) refactoring to decrease system complexity, volume, and program effort, (3) identifying process modules between process variants for systematic reuse, and finally, (4) identifying and developing core functionality in the case of mechatronic variants.

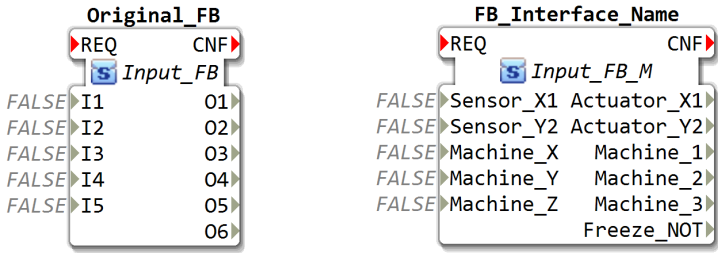
Based on the described work, it is evident that process and mechatronic functionality can be expressed through various approaches, involving different types of encapsulation levels, granularity levels, and FB types. Achieving the right balance among these factors is crucial to express the process functionality, ensuring that the overall control software remains maintainable as the number of variants increases.

3 Modularization Guidelines in IEC 61499

We describe six modularization guidelines derived from insights gained through the analysis of industrial control software developed in the IEC 61499 standard. The different guidelines are based on: (1) the proper naming of FB interface elements, (2) structuring FB based on variants, (3) positioning FBs based on execution order inside an application, (4) encapsulating process functionalities, (5) choosing the correct encapsulation level, and finally, (6) avoiding over-modularization.

3.1 Guideline 1: Interface Naming

Legacy control software often consists of FBs with short names for interface elements, i.e., *Input (I)* and *Output (O)*. The FB interface elements are named *In* and *On*, where *n* represents the number of interface elements. For example, if you have an *Input_FB* with five inputs and six outputs, you would have interface elements labeled *(I1 to I5)* for inputs and *(O1 to O6)* for outputs. The naming of interface elements is done in the manner mentioned due to the legacy system tool support and is often followed by a comment for each interface element. These comments can contain information about the usage of interface elements, for instance, the source and destination of the interface elements.



(a) Original FB. (b) Original FB modified.

Fig. 1: Guideline 1 - Interface Naming.

The clarity of FB usage is compromised when the interface is given an ambiguous name. While IEC 61499 allows comments for interface elements, we recommend users to appropriately name the FB interface based on the source and destination for input and output interface elements. Fig. 1a shows the original FB from legacy software, whereas Fig. 1b is the same original FB, but modified with the proper interface names.

Input_FB_M is a modified version of *Input_FB* with a proper interface name based on the input it receives and the output it sends. *Input_FB_M* receives input from Sensors X1 and Y2, and Machines X, Y, and Z, and outputs data to Actuators X1 and Y2, Machines 1, 2, and 3, and Freeze_NOT, respectively.

We encourage users to align the names of the FB interface with the source and destination of input and output interface elements. Proper interface naming guides the developer in using the FB correctly, as chances of misconnection are reduced due to a well-defined source and destination.

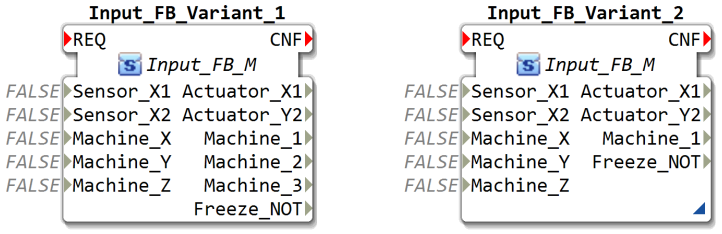
3.2 Guideline 2: FB Structure

FBs in the IEC 61499-based tool, Eclipse 4diac IDE[2], have a feature allowing users to set pins' visibility. Fig. 2a shows the input FB for *Variant 1* and it has output interfaces required for the Actuators X1 and Y2 and Machines 1, 2, and 3 and Freeze_NOT. In contrast, *Variant 2*, as shown in Fig. 2b, has output interfaces required for Actuator X1, Y2, Machine 1 and Freeze_NOT. Since the output interfaces for Machines 2 and 3 are not used, the developer hides the interface elements *Machine_2* and *Machine_3* of the *Input_FB_M* as seen in Fig. 2b. The FB with hidden interfaces contains a *Hidden Pin Marker* at the bottom right in blue as shown in Fig. 2b.

Hiding FB interfaces may lead to three effects. Firstly, both variants might contain the same algorithm, resulting in unused extra code for *Variant 2*, resembling the *Unused Data* bad

[2] https://eclipse.dev/4diac/

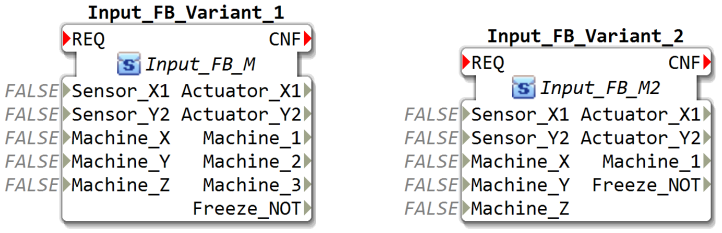
smell [So21]. This *Unused Data* might be due to the absence of process functionalities in *Variant 2*. Secondly, locking the unused code for *Variant 2* with parameters can complicate the maintenance process and, thus, is not desired for variability management [Sh23b]. Thirdly, hiding FB interfaces proves suboptimal for managing variability, requiring additional effort for engineers to identify all variant cases and document which interface elements should be hidden.



(a) Input FB Variant 1. (b) Input FB Variant 2.

Fig. 2: Guideline 2 - FB Structure with hidden interfaces.

To mitigate these possible effects of the FB in Fig. 2, we recommend creating separate FBs for each variant. Fig. 3 depicts the revised FBs, with separate FBs for *Variant 1* and *Variant 2*, each having their own algorithms. Unused interface elements in *Variant 2* are removed rather than hidden, addressing the issue of *Unused Data*. This approach also eliminates the necessity for parameters to lock unused functionality, resulting in an improved FB structure from a variability management perspective.



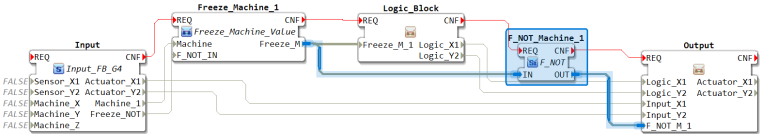
(a) Explicit Input FB Variant 1. (b) Explicit Input FB Variant 2.

Fig. 3: Guideline 2 - Improved FB Structure.

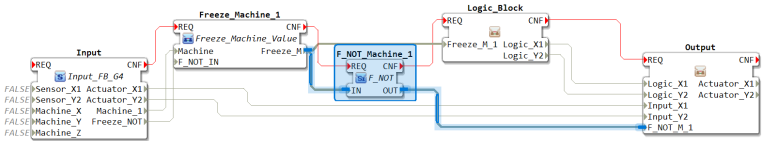
The results from Guideline 2 promote the development of explicit FBs. These FBs, designed based on the variants, serve as the initial solution to address the three side effects of hiding interface elements. As a result, the FBs have an improved structure with well-defined interface elements for both variants.

3.3 Guideline 3: FB Execution order in the FB Network

In modularization, an irregular distribution of functionalities across FBs within an application can lead to challenges in the maintenance process. Thus, we recommend to ensure that related FBs are appropriately placed together based on the execution order. This will promote logical organization of FBs and execution of functionalities inside an application. For instance, Fig. 4a depicts FB *F_NOT_Machine_1* which is placed between *Logic_Block* and *Output*. FB *F_NOT_Machine_1* receives data from *Freeze_Machine_1* and sends its output data to *Output* when an event is triggered. Apart from the above data interaction, FB *F_NOT_Machine_1* is independent of other FBs. Fig. 4b represents reordered *F_NOT_Machine_1* after *Freeze_Machine_1* in the FB network. Such reordering is advantageous as it brings FBs corresponding to certain functionalities together without affecting the overall process functionality. The same can be followed for other FBs in the application if they belong together with similar data interaction.



(a) FB network with functionality spread.



(b) FB better structuring.

Fig. 4: Guideline 3 - FB Execution Order in FB Network.

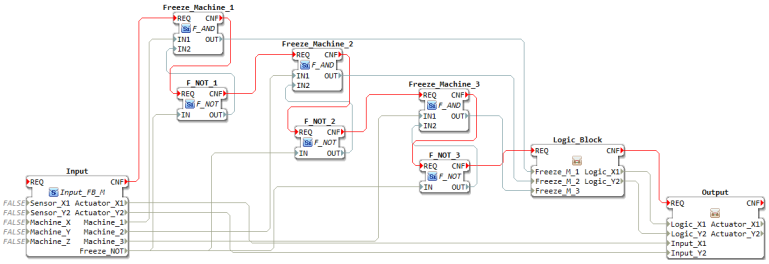
Reordering allows to group FBs by functionality and define variation points, ultimately reducing functionality spread in the application. This organization is vital from the perspective of variability management, facilitating a more structured and streamlined system.

3.4 Guideline 4: What to encapsulate from the FB Network?

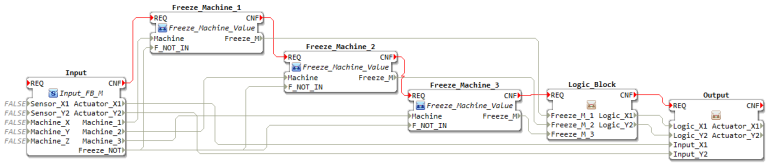
When components are not properly encapsulated, making changes to one component may inadvertently impact others, making variability management and maintenance more complex. To address this challenge, we recommend identifying control software components consistently used together in all variants. Then, they can be encapsulated into more meaningful types that represent process functionality. Fig. 5a displays three instances where the combination of *F_AND and F_NOT* are used in the FB network. The combination

of FBs can be encapsulated together as they always exist together and similarly interact with each other through data and event connections. Encapsulating the FBs into a module will provide an increased overview, which can be saved in the type library and used in all variants [Sh23b].

We compose the FB *F_AND and F_NOT* inside a subapplication and save it as a type *Freeze_Machine_Value* in the type library. Fig. 5b represents the FB network after encapsulating it as a *Typed Subapplication*. If FBs are encapsulated within a subapplication, their hierarchy level is reduced by one. The encapsulated FBs are not accessed directly, but via interfaces of the subapplication. Overall, the visual complexity of the network is reduced with the use of three instances of *Freeze_Machine_Value* in Fig. 5b compared to the FB combination in Fig. 5a.



(a) FB network before Encapsulation.



(b) FB network after Encapsulation.

Fig. 5: Guideline 4 - What to encapsulate from FB Network?

We recommend identifying control software components consistently used together in all variants and encapsulate them as modules. Guideline 4 reduces the overall visual complexity of the FB network while leaving the process functionality unchanged.

3.5 Guideline 5: Encapsulation Level

Developers can achieve functional consistency by utilizing a range of encapsulation levels. This may be useful when the developer to seeks to keep the same hierarchy level, for instance, based on company directions. To encapsulate without changing the hierarchy level, the developer can choose different available FB types that can be used at the same hierarchy level. Different encapsulation levels, *Subapplication*, *Typed Subapplication* and

Simple FB, to represent the same process functionality are shown in Fig. 6. For example, *Freeze_Machine_Value* is a new FB of type *Simple*, which can perform the same functionality as *Typed Subapplication Freeze_Machine_Value*, all while maintaining the same hierarchy level.

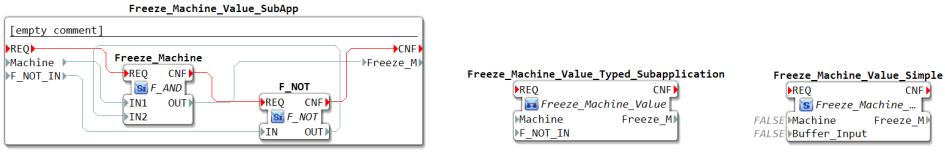


Fig. 6: Guideline 5 - Different Encapsulation Level.

Guideline 5 is practical when developers want to keep the same hierarchy of the FBs. It is an alternative to introducing a new hierarchy as described in Guideline 4. Both approaches result in the creation of new types in the type library with different levels of encapsulation and serve as alternatives to the FB combination (F_AND and F_NOT).

3.6 Guideline 6: Don't Over-Modularize

After examining the approaches for organizing control software into modules, it is pertinent to explore how this can be efficiently achieved without falling into the trap of excessive modularization. Fig. 7 depicts an example of an over-modularized control software, where three variant-based modules are created, which use different instances of *Freeze_Machine_Value*. Specifically, *Freeze_Machine* uses one instance, *Freeze_Machine_1_2* uses two instances connected in series, and *Freeze_Machine_1_2_3* uses three instances connected in series within the subapplication, as shown in Fig. 8. Later, the subapplications are converted to *Typed Subapplications*. These typed subapplications are named based on the variant, and interface names are provided to interact with the *Freeze_Machine_Value* modules. However, this encapsulation results in three separate modules that need to be maintained. In the future, if modifications are required, the developer must make changes in three modules, *Freeze_Machine_Value*, *Freeze_Machine_1_2* and *Freeze_Machine_1_2_3*, separately in Fig. 7.

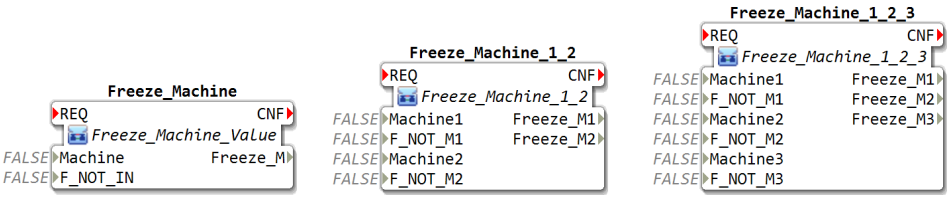


Fig. 7: Guideline 6 - Example of Over-Modularization based on variants.

Instead, we recommend using subapplications when multiple instances of the same module are required for variants. Fig. 8 demonstrates the same functionality as the three modules created through over-modularization in Fig. 7. In the future, if modifications are required, the developer only needs to make changes in one module, i.e., *Freeze_Machine_Value*, and those changes will be applied automatically to all three subapplications in Fig. 8.

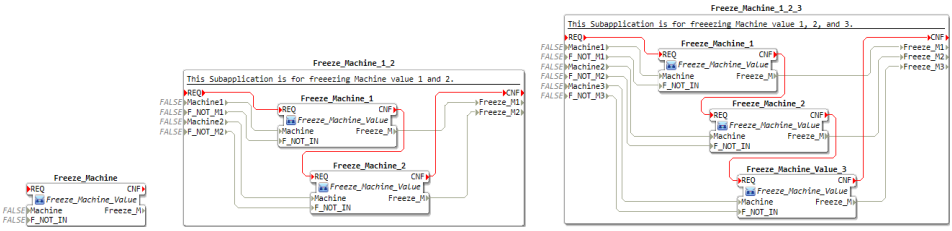


Fig. 8: Guideline 6 - Subapplication instead of creating types for every variant.

Do not create a module as a typed subapplication if the variants require multiple instances of the same module. Instead, use a subapplication with the proper name to compose the required instances for the variant. Utilizing subapplications will help prevent type explosion in the type library as the number of variants increases.

4 Conclusion

We introduced six initial guidelines to improve the control software modularization for better managing variability. *Guidelines 1 and 2* showcased the importance of having a proper interface name and explicit FB based on variants. While having variant-based explicit FBs can increase the types in the type library, it offers a favorable trade-off compared to using the same FB with an algorithm locked by parameters. *Guideline 3*, which allows the restructuring of the FB within the application, serves as a starting point for identifying process-related variation points. Alternatively, we found that encapsulating functionalities as *Typed Subapplications* in *Guideline 4* or upgrading to a new type, as suggested in *Guideline 5*, reduced the visual complexity of the application. Finally, *Guideline 6* provided insights on avoiding over-modularization in control software.

Next, we will gather feedback on the developed guidelines from our industrial partner. Currently, we are examining the quality of legacy software architecture with and without these modularization guidelines. While the presented modularization guidelines primarily focus on managing variability and result in the creation of variant explicit modules, we anticipate that they may increase maintenance efforts, as they are not specifically maintenance-focused. In our future work, we will also aim to achieve a balance between maintenance and variability management.

References

[Ba23] Bauer, Philipp; Sonnleithner, Lisa; Rabiser, Rick; Zoitl, Alois: Using Graph Clustering Algorithms to Modularize IEC 61499 Control Software. In: 2023 IEEE 28th Intl. Conf. on Emerging Technologies and Factory Automation (ETFA). IEEE, pp. 1–4, 2023.

[Be20] Berger, Thorsten; Steghöfer, Jan-Philipp; Ziadi, Tewfik; Robin, Jacques; Martinez, Jabier: The state of adoption and the challenges of systematic variability management in industry. Empirical Software Engineering, 25:1755–1797, 2020.

[Ch11] Chiriac, Noemi; Hölttä-Otto, Katja; Lysy, Dusan; Suk Suh, Eun: Level of modularity and different levels of system granularity. 2011.

[DB15] Dörbecker, Regine; Böhmann, Tilo: Tackling the Granularity Problem in Service Modularization. In: AMCIS. 2015.

[LM96] Lampel, Joseph; Mintzberg, Henry: Customizing customization. MIT Sloan Management Review, 1996.

[LYL14] Lee, Dong-Ah; Yoo, Junbeom; Lee, Jang-Soo: Guidelines for the Use of Function Block Diagram in Reactor Protection Systems. In: 2014 21st Asia-Pacific Software Engineering Conf. volume 1, pp. 135–142, 2014.

[ME98] Miller, Thomas D; Elgard, Per: Defining modules, modularity and modularization. In: Proc. of the 13th IPS research seminar, Fuglsoe. Aalborg Universiy Fuglsoe, 1998.

[Pa18] Patil, Sandeep; Drozdov, Dmitrii; Zhabelova, Gulnara; Vyatkin, Valeriy: Refactoring of IEC 61499 function block application—a case study. In: 2018 IEEE Industrial Cyber-Physical Systems (ICPS). IEEE, pp. 726–733, 2018.

[RZ21] Rabiser, Rick; Zoitl, Alois: Towards mastering variability in software-intensive cyber-physical production systems. Procedia Computer Science, 180:50–59, 2021.

[Sh23a] Sharma, Shubham; Fadhlillah, Hafiyyan Sayyid; Fernández, Antonio M Gutiérrez; Rabiser, Rick; Zoitl, Alois: Modular Control Software Design to Support Mechatronic Variants in IEC 61499. In: 2023 IEEE 28th Intl. Conf. on Emerging Technologies and Factory Automation (ETFA). IEEE, pp. 1–8, 2023.

[Sh23b] Sharma, Shubham; Fadhlillah, Hafiyyan Sayyid; Gutiérrez Fernández, Antonio Manuel; Rabiser, Rick; Zoitl, Alois: Modularization Technique to Support Software Variability in Cyber-Physical Production Systems. In: Proc. of the 17th Intl. Working Conf. on Variability Modelling of Software-Intensive Systems. VaMoS '23, Association for Computing Machinery, New York, NY, USA, p. 71–76, 2023.

[So21] Sonnleithner, Lisa; Oberlehner, Michael; Kutsia, Elene; Zoitl, Alois; Bácsi, Sándor: Do you smell it too? Towards Bad Smells in IEC 61499 Applications. In: 2021 26th IEEE Intl. Conf. on Emerging Technologies and Factory Automation (ETFA). pp. 1–4, 2021.

[So22] Sonnleithner, Lisa; Bauer, Philipp; Rabiser, Rick; Zoitl, Alois: Applying visualization concepts to large-scale software systems in industrial automation. In: 2022 Working Conf. on Software Visualization (VISSOFT). IEEE, pp. 182–186, 2022.

[Zo14] Zoitl, Alois; Lewis, Robert et al.: Modelling control systems using IEC 61499. Technical report, IET, 2014.

Student Research Competition

Message from the SE'24 Student Research Competition Chairs

Leif Bonorden[1], Sören Henning[2]

Abstract: The Software Engineering 2024 conference (SE'24) features a Student Research Competition. This volume includes summaries of the nominated student works.

1 The SE'24 Student Research Competition

The Student Research Competition takes place as part of the SE'24 program. It aims to give students a forum to present their results at a conference and to establish contacts with researchers and practitioners from the German-speaking software engineering community.

We called on students to submit summaries of their bachelor's or master's theses on software engineering topics for the SE'24 Student Research Competition. The summary should have outlined the topic, motivation, objective, implementation, and results of the thesis. All works that were completed after January 1, 2022, and have not already been submitted to the Student Research Competition of an earlier SE edition were eligible.

All submissions received were assessed for their relevance to software engineering research. From these, we made a pre-selection of contributions that were invited to present their work with a poster and a short talk at SE'24 conference. An expert jury with representatives from academia and industry awards prizes to the best submissions based on the evaluation of the written contributions, the poster, and the presentation.

2 Nominated Student Theses

In total, we nominated seven students to present their works at the SE'24 Student Research Competition. We would like to point out that all received submissions are of excellent quality. The following bachelor's and master's theses were nominated and are included as summaries in this volume:

- Ulrike Engeln: *Code Smell Detection using Features from Version History*

[1] Universität Hamburg, Germany, leif.bonorden@uni-hamburg.de
[2] Johannes Kepler University Linz, Austria, soeren.henning@jku.at

doi:10.18420/sw2024-ws_12

- Robin Kimmel: *Large Language Models for Engineering Web Applications*
- Nathan Hagel: *Modeling and Simulation of Dynamic Containerized Software Architectures using Palladio*
- Maximilian Kreb: *Prädiktive, statische Energieverbrauchsanalyse basierend auf experimentell ermittelten Energiemodellen*
- Thomas Larcher: *CORE: Code Once, Run Everywhere. Engineering Serverless Workflow Applications with High-Level of Abstraction*
- Jingxi Zhang: *Towards the Transformation of heterogeneous Language Components*
- Niklas Krieger: *HyLiMo: A Textual DSL and Hybrid Editor for Efficient Modular Diagramming*

3 Acknowledgments

We would like to thank all students who submitted their works as well as their supervisors and faculty for encouraging students to submit their work.

Moreover, we would like to thank the other members of the jury for evaluating the nominated student theses.

A special thanks goes to the organizing committee of the SE'24 who made it possible to organize the Student Research Competition as part of the supporting program of the SE'24.

Hamburg, Linz, February 2024
Leif Bonorden, Sören Henning

Code Smell Detection using Features from Version History

Ulrike Engeln[1]

Abstract: Code smells are indicators of bad quality in software. There exist several detection techniques for smells, which mainly base on static properties of the source code. Those detectors usually show weak performance in detection of context-sensitive smells since static properties hardly capture information about relations in the code. To address this information gap, we propose a strategy to extract information about interdependencies from version history. We use static and the new historical features to identify code smells by a random forest. Experiments show that the introduced historical features improve detection of code smells that focus on interdependencies.

Keywords: code smells detection; machine learning; mining sofware repositories

Code smells are structures in code that commonly require re-engineering. In literature, there exist several attempts to automate the time-consuming and costly process of code smell detection. Most of them are based upon static properties of the software, e. g., code metrics. However, not all design flaws are of structural nature. There exist three different kinds of code smells. They target *coding style*, *responsibilities*, and *interdependencies* of classes or methods. While smells that target *coding style* are of structural nature only, smells that describe *interdependencies*, e. g., Feature Envy, cannot be detected by structural properties. For identification of such smells, Palomba et al. [Pa15] propose using the version history of the code as source of information. Smells targeting *responsibilities* affect both, code metrics and version history. For example, God Classes usually have many lines of code and appear frequently in the version history.

Machine learning techniques that learn based on features, e. g., code metrics, are well suited for the development of a smell detector. To enable a classifier to detect all three kinds of smells, Barbez et al. [BKG20] introduce a hybrid, ensemble learning-based smell detection technique, which combines classifiers using code metrics and information from version history. We propose a different approach of combining the two sources of information, which, rather than combining existing detectors, directly learns identification of smells from the two sources of information.

Since in their work Palomba et al. do not draw features from version history but apply heuristic rules, one major issue of our machine learning approach is to decide how to express information from the version history by features. The introduced method of feature extraction from version history builds the core of our work. Figure 1 illustrates our general idea of feature extraction. We measure how often files or methods change simultaneously.

[1] Hamburg University of Technology, Institute for Software Systems ulrike.engeln@tuhh.de

doi:10.18420/sw2024-ws_13

We introduce three design parameters to determine which parts of version history are considered as simultaneous: orientation, window size, and weighting. For each file, we create a vector containing entries for all available files. We identify all commits containing the file and sum up the weights of all files that appear within the defined window (here: forward oriented of size 4 starting from commit 2). As historical features we use the statistical description of the normalized weight vectors.

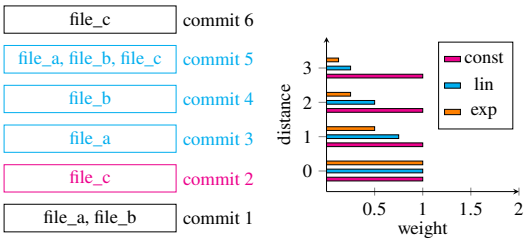


Fig. 1: Concept for distributing weights from git history for *file_c*. Relevant commits are colored. Right: different weighting concepts.

By experiments, we investigate whether using metrics and historical features allows for better code smell detection than using only code metrics. We evaluate the code smells God Class and Feature Envy, since we expect smells targeting *responsibility* and *interdependencies* to show in version history. We cross-validate a random forest of 100 decision trees based on data from previous work [ML20; Pa15]. The experiments show that historical features contain information about code smells. The main results are given in Table 1. For God Class, we observe improvements of only 0.3% to 1.8% in all performance measures but precision. For Feature Envy, improvements are significant, e. g., F1-score increases from 43.03% to 54.36%. Those results show that historical features allow for better detection of smells targeting *interdependencies*, while for *responsibilities*, the observed improvement is too small to assume an impact.

Tab. 1: Performances with and without historical features.

	Feature Envy		God Class	
	code metrics	with history	code metrics	with history
Accuracy	90.79%	92.18%	95.95%	96.22%
F1-Score	43.03%	54.36%	74.87%	75.72%
Precision	57.20%	70.20%	77.22%	76.83%
Recall	42.37%	49.94%	73.96%	75.74%
AUC	0.6885	0.7323	0.8651	0.8737

References

[BKG20] Barbez, A.; Khomh, F.; Guéhéneuc, Y.: A machine-learning based ensemble method for anti-patterns detection. J. Syst. Softw. 161/, p. 110486, 2020.

[ML20] Madeyski, L.; Lewowski, T.: MLCQ: Industry-Relevant Code Smell Data Set. In. EASE '20, ACM, Trondheim, Norway, pp. 342–347, 2020.

[Pa15] Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; de Lucia, A.: Mining Version Histories for Detecting Code Smells. IEEE Trans. Softw. Eng 41/5, pp. 462–489, 2015.

Large Language Models for Engineering Web Applications

Robin Kimmel[1]

Abstract: This work examines the potential of blending traditional programming methods with artificial intelligence, specifically large language models (LLMs), to automate the creation of web applications. The primary focus is on defining the necessary software architecture and components to transform diverse inputs, such as natural language and Unified Modeling Language (UML) notations, into functional web applications. The core concept involves a software agent built around a Large Language Model, equipped with tools to autonomously address tasks. While demonstrating promise, this approach exhibits certain limitations that demand further exploration and refinement.

1 Fundamentals

Large language models utilize billions of parameters and are trained with large quantities of mostly natural language text. They are successfully used in a variety of language-based tasks like summarization, translation, text generation and more [Mi23]. The behavior of LLMs can be engineered using prompts that influence future responses. This is called prompt engineering. It is also possible to fine-tune models by retraining them with specific datasets. Changes to the model through fine-tuning tend to deliver more reliable results than just prompt engineering. However, due to the stochastic nature of LLMs, they may produce falsely formatted or factually false content.

A software agent acts on behalf of a third party or an objective and tries to fulfill this objective as well as possible. Each agent must have a certain amount of autonomy to do this; otherwise, it would simply be a pre-defined instruction list [Gr97]. In the context of this work, a software agent will be a pseudo-personification of a digital assistant that thrives to create a web application according to the user's specifications. The agent can take multiple steps to get closer to its final goal while continuously checking its environment and the effect of its own actions on it.

2 Realisation

The whole application can be separated into three main components: the agent, a parser and the tools. The agent component includes all the functionality needed to get an answer from the LLM. When given a task, the agent will give an answer on what to do to solve the provided objective. The parser then takes this completely text-based answer and tries

[1] University of Stuttgart, ISW, Seidenstr. 36, D-70174 Stuttgart, Germany Robinkimmel98@gmx.de

doi:10.18420/sw2024-ws_14

to convert all the information into a machine-readable format. After that, either a tool is called or the agent thinks it is done and has finished its objective. A tool is basically a function that interacts with the environment on behalf of the agent. All the tools need to generate text-based feedback that can be fed back to the agent in order for him to decide what to do next. For creating web applications, the most important tools consist of read and write functions. This loop of calling the agent, parsing the answer and using a tool can be repeated N times. The UML activity diagram, representing the high-level view of the whole application's lifecycle, is shown in Figure 1.

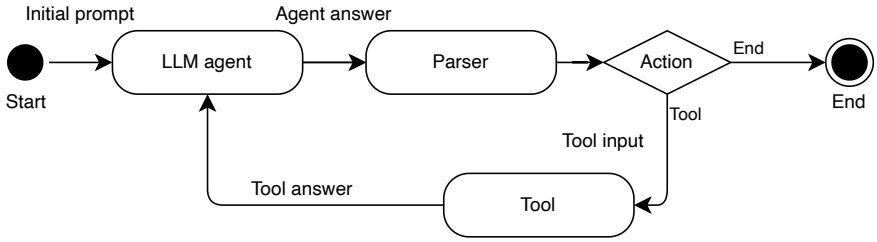


Fig. 1: The UML activity diagram for the agent lifecycle, from when it is started to when it terminates.

The main challenges arising during the implementation of the above-mentioned high-level structure are: creating the prompt engineering for the agent itself, including a schema of what a step includes as well as a format for the text answer, a robust parser that can handle deviations from the desired format and multiple error handling mechanisms to make the whole process more robust.

3 Results

This approach results in an architecture that can create web applications based on the Django framework fully autonomously, solely based on specifications in natural language provided by the user. However, the current implementation is not yet sufficient to be used in real-world applications, but it functions as a proof of concept that this kind of architecture can be used to solve the underlying task. In the future, for example, multi-agent approaches will be introduced to further improve performance.

Bibliography

[Gr97] Green, Shaw; Hurst, Leon; Nangle, Brenda; Cunningham, Padraig; Sommers, Fergal; Evans, Richard: Software Agents: A review. Trinity College Dublin and Broadcom Éireann Research Ltd.2, 1997.

[Mi23] Min, Bonan; Ross, Hayley; Sulem, Elior; Veyseh, Amir Pouran; Nguyen, Thien Huu; Sainz, Oscar; Agirre, Eneko; Heintz, Ilana; Roth, Dan: Recent advances in natural language processing via large pre-trained language models: A survey. ACM Computing Surveys, 56(2):1–40, 2023.

Modeling and Simulation of Dynamic Containerized Software Architectures using Palladio

Nathan Hagel[1]

Abstract: Nowadays, distributed applications are often not statically deployed on virtual machines. Instead, a desired state is defined declaratively. A control loop then tries to create the desired state in a cluster. To predict the impact on the performance of a system using these deployment techniques is difficult. This paper introduces a method to predict the performance impact of the usage of containers and container orchestration in the deployment of a system. Our proposed approach enables system simulation and experimentation with various mechanisms of container orchestration, including autoscaling and container scheduling using the Palladio-Component-Model (PCM). We validated this approach using a Kubernetes reference cluster which we modelled using a workflow defined in the authors bachelor thesis [Ha22] [Ha23]. Our findings suggest, that the most common concepts in container orchestration can be modelled and simulated using Palladio and the PCM-Extension of [Ha22].

Keywords: Container; Performance Prediction; Container Orchestration

1 Motivation

In recent years, container technologies have become increasingly important for software developement and deployment [Zh18]. Beyond the mere use of containers, container orchestration tools like Kubernetes play a significant role. They allow a declarative description of the desired system state, which the container orchestrator then automatically creates and monitors. The question arises, how these technologies impact a software's performance and which scaling and allocation strategies are the most appropriate.
Gaining these insights from real-world experiments can be costly or restrict the user experience. Using the software architecture simulator Palladio [Re16], component-based applications can already be analyzed. Extending it with container and container orchestration concepts can improve early stage performance analysis and simplify the decision-making process regarding a possible containerization of a system or architecture.

2 Goal and Methodology

To simulate dynamic, containerized software architectures using Palladio, we need to be able to model not only the concept of containers but also the main concepts of container orchestration. The selection of these cocepts was based on Kubernetes as the de facto

[1] Karlsruhe Institute of Technology, Chair of Modelling for Continuous Software Engineering (Prof. Dr. Anne Koziolek), Kaiserstraße 12, 76131, Karlsruhe, Germany, nathan@hagel.dev

doi:10.18420/sw2024-ws_15

standard in this area. The mapped concepts are: Clusters, Nodes, Pods, Ressource-Requests and Limits, Services, Ingress, Deployments, Pod-Scheduling and Autoscaling. Besides a mapping to model the concepts, a solution had to be found to analyze dynamic Deployments. After defining which concepts were needed to realistically simulate containerized systems in Palladio, a requirements-analysis was conducted for each concept to determine which properties and capabilities are required for the model and the simulation. Then, the existing PCM-Elements were analyzed to find out where existing concepts could be reused or extended. To implement the dynamic concepts like Pod-Allocation, Autoscaling etc., we looked at model transformations and developed a concept to implement capabilites like automatic Pod-Allocation and a Horizontal-Pod-Autoscaler (HPA). Futhermore, a Pod-Allocation-Scheduler for PCM-Models was implemented and tested. For evaluation purposes, we determined the share of implemented and for the analysis relevant concepts. Additionally, we defined a reference cluster based on an existing application and modeled this cluster to determine the limits of our extension. Finally, the decisions of the implemented Pod-Allocation-Scheduler were compared to the decisions of a standard Kubernetes scheduler.

3 Results

Based on the defined requirements for each Kubernetes' concept, we were able to find or define a mapping for almost all the concepts into the PCM. The only exception is the Replica Set which has a strong overlap with the Deployment definied in our extension and was therefore not mapped. For the implemented Pod-Allocation-Scheduler, tests showed that it behaves the same as the Kubernetes standard implementation. The defined reference cluster could be fully modeled using the workflow to use the PCM-Extension which was also defined in [Ha22]. A solution was proposed to implement the relevant control loops using model reconfigurations under consideration of possible transient effects. In a follow-up project the proposed dynamic simulation concept for this extension is currently implemented.

Literatur

[Ha22] Hagel, N.: Modellierung und Simulation von dynamischen Container-basierten Software-Architekturen in Palladio, Bachelor's Thesis, Karlsruher Institut für Technologie, 2022.

[Ha23] Hagel, N.: Modellierung und Simulation von dynamischen container-basierten Software-Architekturen in Palladio. In: Proc. 25. Workshop Software-Reengineering Evoluation (WSRE 2023). S. 22–23, 2023.

[Re16] Reussner et al.: Modeling and Simulating Software Architectures - The Palladio Approach. 2016, ISBN: 9780262034760.

[Zh18] Zhang et al.: A Comparative Study of Containers and Virtual Machines in Big Data Environment. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). S. 178–185, 2018.

Prädiktive, statische Energieverbrauchsanalyse basierend auf experimentell ermittelten Energiemodellen

Maximilian Krebs[1]

Abstract: Die Bestimmung des Energieverbrauchs moderner Rechensysteme stellt Entwickler:innen vor neue Herausforderungen. Bisherige Verfahren verlangen komplizierte Messaufbauten und lassen sich schwer in bestehende Workflows einbetten. Im Rahmen meiner Bachelorarbeit wurde daher ein Tool entwickelt, dass den Energieverbrauch eines Eingabeprogramms statisch approximiert und für die ausführende CPU beschreibt. Der Prozessor wird dazu mithilfe von Intel RAPL und LLVM auf seinen charakteristischen Energieverbrauch untersucht. Eine Analyse berechnet anschließend näherungsweise auf Grundlage der charakteristischen Energiewerte die Energie, die das Eingabeprogramm verbrauchen würde. Die Evaluation der bestimmten Energiewerte zeigt das Potenzial der statischen Analyse des zu erwartenden Energieverbrauchs, legt aber nahe, dass das verwendete Programmmodell noch erweitert und verbessert werden sollte.

Keywords: LLVM, Intel RAPL, Energieverbrauch, Statische Analyse, Green Coding

1 Statische Analyse des zu erwartenden Energieverbrauchs

Die statische Analyse des zu erwartenden Energieverbrauchs eines Programms stellt eine Möglichkeit dar, um Entwickler:innen direkt im Entwicklungsprozess mit Metriken zum Energieverbrauch ihres Programms zu versorgen. Eine Entwicklung hin zu einer energiebewussteren Programmierung stellt dabei im Kontext des Klimawandels eine sinnvolle Erweiterung bisheriger Analyseaspekte dar. Zur Bestimmung des Energieverbrauchs wurde im Rahmen meiner Bachelorarbeit eine Analysepipeline entwickelt, die unter Aufteilung in einen Profiling- und in einen Analyse-Teil den Energieverbrauch eines Eingabeprogramms approximiert. Mithilfe von LLVM wurde dabei ein Compiler Pass entwickelt, der durch die Entwicklung weiterer Tools in gängige Programmierumgebungen wie z.B. Visual Studio Code eingebettet werden kann.

2 Ablauf der Analyse

Der Energieverbrauch eines Programms kann letztlich auf den Energieverbrauch der einzelnen Instruktionen zurückgeführt werden. Dabei unterscheiden sich die Kosten der Instruktionen je nach verwendeter Hardware. Damit eine Analyse unabhängig vom zugrundeliegenden Setup durchgeführt werden kann, baut die hier vorgestellte Methode auf einer Profiling-Phase auf, bei der die Kosten der einzelnen Instruktionen quantitativ bestimmt

1 Technische Universität Dortmund, Informatik, Otto-Hahn-Straße 14, 44227 Dortmund, Deutschland, maximilian.krebs@cs.tu-dortmund.de

doi:10.18420/sw2024-ws_16

werden. Dafür werden die einzelnen Instruktionen des x86-Instruktionsets mithilfe von LLVM in unterschiedliche Gruppen eingeteilt. Die Gruppen wurden dabei so gewählt, dass diese sich durch die im Prozessor verrichtete Arbeit unterscheiden. Zu jeder dieser Gruppen wurde eine x86 Instruktion als Repräsentant ausgewählt. Die einzelnen Repräsentanten werden auf der CPU wiederholt ausgeführt. Mithilfe einer Mittlung wird dann der charakteristische Energiewert der zugehörigen Gruppe bestimmt. Die Energiewerte der Gruppen bilden zusammen das Energieprofil des ausführenden Rechners. Die Analyse eines Eingabeprogramms nutzt dieses Profil, um den Energieverbrauch der verwendeten Instruktionen abzuschätzen und um so den Energieverbrauch des Programms im Worst-, Average- und Bestcase zu beschreiben. Grundlage dafür ist ein auf Basis von LLVM entwickelter Compiler-Pass, der die Energiewerte der einzelnen Instruktionen miteinander verrechnet.

3 Ergebnisse

Programm	Worst Case	Average Case	Best Case	Mittelwert	Std. Abweichung	Messung
rsa_encrypt	1, 3349	0, 66188	0, 35362	0, 78347	0, 40973	0, 18668
aeskeywrap	0, 86682	0, 19204	0, 15357	0, 40414	0, 32754	0, 1375
hmac_sha512	0, 4613	0, 19254	0, 16688	0, 27357	0, 13315	0, 12156
x25519	3, 62504	1, 00986	0, 25681	1, 63057	1, 44342	0, 13166
EVP_RSA_keygen	1, 07166	0, 16083	0, 10932	0, 44727	0, 44201	15, 4282

Tab. 1: Vergleich der Analyseergebnisse mit einer Vergleichsmessung für Demoprogramme von OpenSSL. Angaben in Joule

Die Analyse kann unter gewissen Beschränkungen Aussagen über den Energieverbrauch eines Programms bereitstellen. Tabelle **??** zeigt die Ergebnisse der Analyse für Demoprogramme der Kryptografie Bibliothek OpenSSL 2Die Ergebnisse der einzelnen Teilanalysen werden dabei mit einer Vergleichsmessung gegenübergestellt. Der direkte Vergleich zeigt, dass die bestimmte Energie relativ nah an dem gemessenen Energieverbrauch liegt. Bei näherer Betrachtung und insbesondere im Vergleich zu den Ergebnissen der Analyse für einfache Programme (Sortieralgorithmen, Suchalgorithmen etc.) fällt jedoch auf, dass die Analyse nicht alle relevanten Teile der Programme betrachtet. Bestimmte Kontexte wie Aufrufe von Bibliotheksfunktionen und Speicherverwaltung führen zu einer Verzerrung der Ergebnisse. Ungenauigkeiten in den Energieprofilen, die höchstwahrscheinlich auf Störimpulse des umliegenden Systems zurückgeführt werden können, führen bei der Analyse von Schleifen und komplexeren Kontrollstrukturen ebenfalls zu Ungenauigkeiten.
Insgesamt kann die Analyse den Energieverbrauch von Programmen für Entwickler:innen greifbarer machen. Die festgestellten Ungenauigkeiten führen aber zu einer Verzerrung der bestimmten Messwerte und erlauben keine gesicherte Aussage über die verbrauchte Energie eines Programms. Zukünftige Arbeit in diesem Bereich könnte jedoch die zugrundeliegende Pipeline verbessern und erweitern, um verlässlichere Energiewerte zu liefern.

2 Sourcecode der betrachteten Programme: https://github.com/openssl/openssl

CORE: Code Once, Run Everywhere. Engineering Serverless Workflow Applications with High-Level of Abstraction

Thomas Larcher, University of Innsbruck, Austria

This **master thesis** is supervised by Ass. Prof. Sashko Ristov and defensio is on 30.10.2023.

Abstract: To keep the serverless functions lightweight, a significant portion of the computing is typically offloaded to already pre-trained AI-based Backend-as-a-Service (BaaS) cloud services, such as speech recognition. The recent rise of federated serverless computing offers cost and performance advantages for these *BaaS-enabled serverless workflows* by deploying them across different cloud providers. However, due to the lack of interoperability among cloud providers, many challenges remain to setup the BaaS-enabled serverless workflows in federated serverless infrastructures.

Contribution. To bridge these gaps, this master thesis introduces *CORE* – a novel framework to code portable functions with interoperable BaaS services in a uniform manner, including text-to-speech, speech-to-text, translate, and OCR, together with a globally-federated storage infrastructure, comprising AWS and Google cloud providers *CORE*'s programming model introduces a novel concept of *ServerlessIntents*, which abstract BaaS services from different providers. The *CORE* SDK, supported by *CORE* runtime heuristics and mechanisms, enables dynamic selection of BaaS services during runtime, without the need to rewrite and redeploy the serverless functions. Finally, the *CORE* platform provides a novel mathematical model and a scheduler that selects where to run workflow functions and which BaaS services should be used to optimize their performance.

Experimental results. CORE improves lines of code, maintainability index, and workflow execution time by up to 94.87 %, 41.27 %, and 57.87 %, respectively.

***CORE* programming model** defines ServerlessIntent (Fig. 1), which contains *intentInput* and *intentOutput*, represented with values for location and the way data is passed. The main novelty is the set of *intentFeaturesScope* that can be selected, regardless of which provider supports them. For each *intentFeature*, developer can code the values, which also may be supported by some providers.

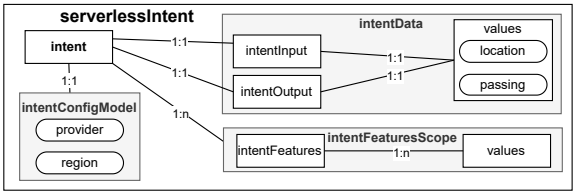


Abb. 1: ServerlessIntent structure.

***CORE* runtime mechanisms** (Fig. 2) support the full life-cycle of the ServerlessIntent: *developer APIs* of the *CORE* programming model that are exposed to developers to code the ServerlessIntent; *provider selector* to dynamically determine which implementations of the BaaS service to call; *input adapter* to adapt the interoperable input into the inputs

doi:10.18420/sw2024-ws_17

recognized by the SDKs of the selected BaaS service implementations, *BaaS invoker* to call the selected BaaS service implementations, and *output adapter* to adapt the output of the executed BaaS services into the interoperable output of the ServerlessIntent. *CORE* moves data between storages to allow users to use e.g., AWS Transcribe from Google Storage.

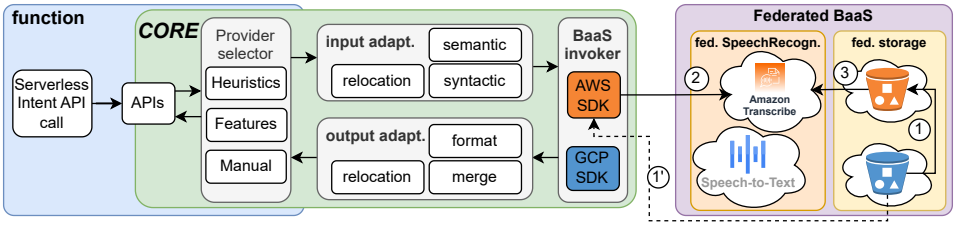


Abb. 2: *CORE* SDK with runtime mechanisms.

***CORE* platform** (Fig. 3) has two internal representations of serverless workflows. *Abstract serverless workflow* (Fig. 3 left), contains all the functions, BaaS services, and data flow dependencies according to the *CORE* model but attaches them to no BaaS and storage backends. *Deployed serverless workflow* (Fig. 3 right) enhances the abstract serverless workflow with the BaaS and storage attachments configured to each function deployed on a cloud region by the *CORE* scheduling algorithm.

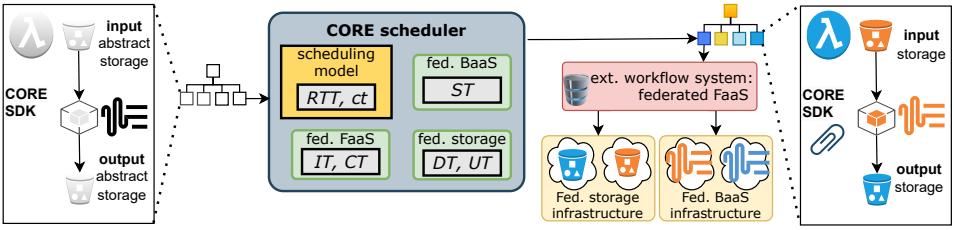


Abb. 3: Overview of *CORE* platform.

Initially, a function developer uses the *CORE* BaaS and storage attachment SDK to attach an *abstract BaaS and storage* to each function. While coding, the function developer does not need to know the underlying BaaS and storage details, such as the cloud region, storage provider, or provider SDK, as *CORE* abstracts them away and provisions them during runtime. In the second step, the developer composes the functions into a serverless workflow through control and data flow dependencies, based on the *CORE* model. *CORE* is workflow composition language and execution engine agnostic. After composing the abstract serverless workflow, the developer forwards it to the *CORE* scheduler for deployment onto concrete computational regions, together with BaaS service and storage attachments.

Towards the Transformation of Heterogeneous Language Components

Jingxi Zhang[1]

Abstract: The expanding number of language workbenches (LWB) has opened up numerous ways of developing domain-specific languages (DSL). Essential components of a DSL are grammar, generator and validation rules. However, the diversity of these DSLs presents challenges when composing language components across LWBs. Our work is dedicated to connecting these disparate language components into a homogeneous DSL. With our transformation concept, we propose a novel way to transform DSLs and demonstrate its practicality on XText and MontiCore, while highlighting encountered challenges and valuable insights. Our work advances the understanding of composing heterogeneous language components by providing a framework for transforming DSLs across the boundaries of LWBs.

1 Motivation

To enhance the collaboration among stakeholders, exhibiting preferences for distinct Language Workbenches (LWBs), the reuse of domain-specific languages (DSLs) serves as a pivotal strategy. Key components of a DSL identified in prior work [Bu20] include validation rules, code generators, and a defined grammar. Within the scope of this work, we delve into a detailed investigation of the transformation of DSL across heterogeneous LWBs, and propose a solution for facilitating a seamless transformation. We provide a demonstration of our concept through the transformation between XText [Be16] and MontiCore [Ru21] DSLs while highlighting inherent challenges.

2 Challenges for Language Component Transformation

The reuse of language components across the boundaries of LWBs necessitate the Abstract Syntax Tree (AST) as an additional component along the trio of components, as it is required by both the generator as well as the validation rules. Therefore, the interconnection between these ASTs is also examined, leading to the following three challenges:
Challenge 1: Feature Analysis of LWBs The first challenge entails an inspection of the chosen LWB pair. This examination is underscored by an analysis of features within the underlying DSL's components.
Challenge 2: Reuse of Generator and Validation Rule Components The second challenge revolves around the implementation phase. By utilizing templates for the target LWB's

[1] University of Stuttgart, Institute for Control Engineering of Machine Tools and Manufacturing Units, Seidenstraße 36, 70174 Stuttgart, Germany jingxi.zhang@isw.uni-stuttgart.de

doi:10.18420/sw2024-ws_18

generator and validation rule, a connection to the source LWB can be established, facilitating the reusability.

Challenge 3: Connecting the Underlying Data Structures The central contribution of this research pertains in establishing a bridge between the ASTs of the LWBs.

3 Developing a Transformation Framework for Language Components

As for the first challenge, in previous investigations [Da19] the transformation of grammars between XText and MontiCore was extensively investigated. Our concept handles the second and third challenges in a generative way. This entails the systematic generation of wrappers around the AST, generator and validation rules, which enables the reuse of the DSL components. As a demonstration, we utilize the XText LWB as the source and the

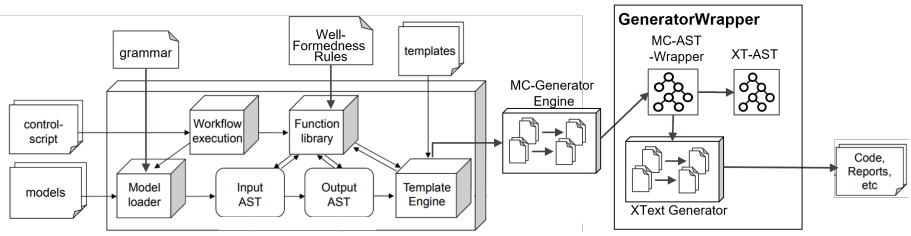


Fig. 1: Workflow after the integration of XText Language Components into MontiCore

MontiCore LWB as the target. We then adapt MontiCore's workflow, depicted in Figure 1, by converting between the ASTs and utilize XText generator and validation rules from wrappers. This allows for a seamless integration of XText components within of MontiCore and ensures the seamless reuse of DSL components.

References

[Be16] Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd, 2016.

[Bu20] Butting, A.; Pfeiffer, J.; Rumpe, B.; Wortmann, A.: A compositional framework for systematic modeling language reuse. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. Pp. 35–46, 2020.

[Da19] Dalibor, M.; Jansen, N.; Kaestle, J.; Rumpe, B.; Schmalzing, D.; Wachtmeister, L.; Wortmann, A.: Mind the gap: lessons learned from translating grammars between MontiCore and Xtext. In: Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling. Pp. 40–49, 2019.

[Ru21] Rumpe, B.: MontiCore Language Workbench and Library Handbook./, 2021.

HyLiMo: A Textual DSL and Hybrid Editor for Efficient Modular Diagramming

Niklas Krieger[1]

Abstract: Diagramming with precise layouting for scientific publications and technical documentations is time-consuming and cumbersome. Therefore, this work briefly presents HyLiMo, a tool for blended graphical and textual diagramming including live-synchronizing. This allows diagrammers to define diagrams textually and then adjust the layout graphically. An evaluation via two case studies confirmed the tool's practicality in creating class diagrams with precise layouts. However, feedback suggests several features for future work.

Keywords: Hybrid diagramming; Graphical-textual diagramming; UML class diagram

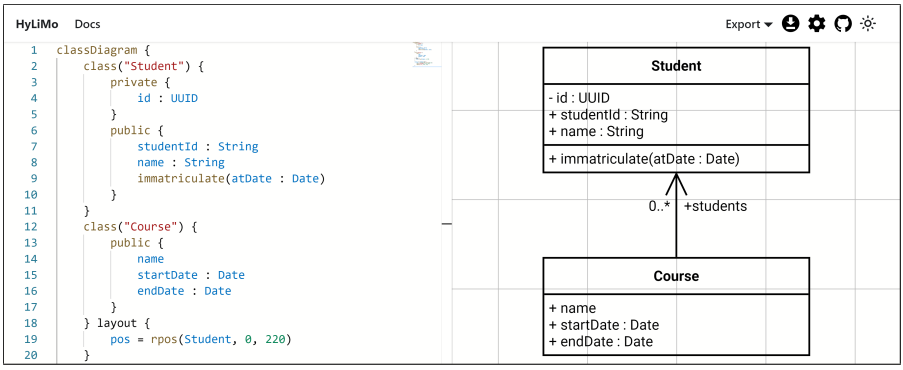


Fig. 1: Web-based hybrid graphical-textual editor.

Introduction and Goals: In Software Engineering, a variety of diagrams are used to create visual representations of systems, processes or concepts. In particular, UML class diagrams are a type of structural diagram used to model classes and their relationships in object-oriented systems. For diagramming, two main approaches exist: First, visual tools like diagrams.net and Visio allow graphically creating diagrams. Users place diagram elements on a canvas, creating the layout of the diagram manually. In contrast, tools like PlantUML and Mermaid are used to create diagrams using a textual syntax. For most such tools, the diagrammar does not specify the layout, instead, elements are auto-layouted. Therefore, this approach is insufficient when manual and precise layouting is required, e.g., for scientific publications and technical documentation. While textual tools supporting manual layouting exist, in particular TikZ UML, defining positions textually can be cumbersome and time-consuming. Hence, users benefit from graphical tools' ability to directly drag diagram elements to the desired location.

[1] University of Stuttgart , Institute of Software Engineering, niklas.krieger@iste.uni-stuttgart.de

doi:10.18420/sw2024-ws_19

Hybrid graphical-textual approaches allow us to bridge this gap. Recent research in modeling introduces the blended modeling concept, which uses multiple notations, including textual and graphical, to manipulate an underlying model [Ci19]. Depending on the task, modelers choose the view most beneficial, thus improving efficiency. Our goal is to bring the benefits of hybrid graphical-textual modeling to diagramming. In particular, users should be able to define diagrams textually initially, but then modify the layout graphically. To achieve this, we envision a hybrid live-synchronized editor, with side-to-side textual and graphical views.

Method: First, we collect and evaluate requirements with different stakeholders. Following, we worked out the concept in detail and created an architecture. After implementing the framework and editor, we evaluated the result in two case studies, as described in *Results*. After collecting initial requirements, to identify further requirements, we conducted expert interviews with 14 current and former software engineering researchers. Based on a survey, where 13 of them participated, we prioritised requirements, and finalized design decisions.

Concept: Fig. 1 shows the hybrid editor[2] created based on these requirements. It is split into two parts: On the left side, the diagram is defined using a textual notation. To improve flexibility and extensibility, we decided to incorporate programming language features, in particular custom functions and control flow constructs. Thus, we decided to implement our textual notation as an internal DSL in a custom general-purpose programming language, SyncScript. The textual notations serve as the single source of truth, including styling and layout information, allowing diagrammers to use tooling made for regular code, in particular version control to store and share diagrams easily. To give the user immediate feedback, textual and graphical views are live-synchronized. On each edit of the textual definition, the code is executed, and the diagram is updated if the execution is successful. As in graphical tools, users can manipulate the layout of the diagram by interacting with the graphical view. In particular, users can move diagram elements, or rotate such elements. On each edit, the textual definition is updated, and as a result, the diagram is rerendered. Yet, in practice, executing the code and generating the diagram introduced lag, worsening user experience. Consequently, we introduced predictions updating the graphical diagram before the code is executed. While the modular architecture of our framework allows us to support different diagram types in the future, currently, only UML class diagrams are supported.

Results: To evaluate our tool, we conducted two case studies in which a doctoral researcher and an undergraduate student created several class diagrams. Used features include custom styling, and a variety of diagram elements, including classes, associations, etc. Both case studies showed that our tool can be used in practice to create class diagrams with precise layouting. However, given feedback includes several feature requests, e.g., pdf export.

Bibliography

[Ci19] Ciccozzi, Federico et al.: Blended Modelling - What, Why and How. In: 2019 ACM/IEEE MODELS Companion. pp. 425–430, 2019.

[2] https://hylimo.github.io/

Autor*innenverzeichnis